FOODIE, Fortran Object oriented Ordinary Differential Equations integration library based on Abstract Calculus Pattern

Zaghi S.[a,1,*], Curcic M.[b,2], Rouson D.[c,3], Beekman I.[c,5], Rossi G.[1,5]

[a] *CNR-INSEAN, Istituto Nazionale per Studi ed Esperienze di Architettura Navale, Via di Vallerano 139, Rome, Italy, 00128*
[b] *Ocean Sciences Rosenstiel School of Marine and Atmospheric Science, University of Miami, 4600 Rickenbacker Causeway Miami, FL 33149-1098 +1 305.421.4000*
[c] *Sourcery Institute 482 Michigan Ave., Berkeley, CA 94707*
[d] *Sapienza, University of Rome, via Eudossiana 18, Rome, Italy*

Abstract

The (numerical) solution of Ordinary Differential Equations (ODEs) problems is of paramount relevance, ODEs system being an ubiquitous mathematical formulation of many physical phenomena (such as those involved in fluid dynamics, chemistry, biology, evolutionary-anthropology, ...). The present paper is the first *manifesto* of FOODIE, a library aimed to numerically solve ODEs problems by means of a clear, concise and efficient *abstract* interface. FOODIE, meaning Fortran Object oriented Ordinary Differential equations Integration Environment, has manifolds aims: to provide a set of built-in numerical schemes that are accurate, robust, validated and efficient and to allow easy application of these schemes to (almost) all ODEs problems by means of an effective Abstract Calculus Pattern. The key idea is to allow the same solver-implementation to be applied to all ODEs problems thus avoiding the re-implementation of the ODEs solver for each different ODEs problem: code re-usability is consequently maximized, FOODIE being a general robust framework. Besides, the same framework also allows rapid development of new ODEs solvers due to the high abstraction level of the library itself.

FOODIE is a modern Fortran library which main features are:

Free FOODIE is a free software;

OOP FOODIE is based on Object Oriented Programming paradigm;

TDD the FOODIE development follows the Test Driven Development software process;

Accurately documented the FOODIE documentation is based on high quality, first class solutions embedding detailed (mathematical) descriptions directly inside code sources; moreover, comprehensive hyper-linked documentations is also provided;

Comphrensive FOODIE provides a comprehensive set of built-in solvers, namely one-step, multi-steps, explicit, implicit, predictor-corrector solvers; moreover, its Abstract Calculus Pattern allows the solution of virtually all kinds of ODEs problems;

Collaborative the development of FOODIE takes advantage of web communications, the main project being hosted on GitHub.

The present paper is the first announcement of FOODIE project: the current implementation is extensively discussed and its capabilities are proved by means of tests and examples.

Keywords: Ordinary Differential Equations (ODE), Partial Differential Equations (PDE), Object Oriented Programming (OOP), Abstract Calculus Pattern (ACP), Fortran

PROGRAM SUMMARY

Manuscript Title: FOODIE, Fortran Object oriented Ordinary Differential Equations integration library based on Abstract Calculus Pattern
Authors: Zaghi, S., Curcic, M., Rouson, D., Beekman, I.
Program title: FOODIE
Journal Reference:
Catalogue identifier:
Licensing provisions: GNU General Public License (GPL) v3
Programming language: Fortran (standard 2008 or newer); developed and tested with GNU gfortran 5.2 or newer
Computer(s) for which the program has been designed: designed for shared-memory multi-cores workstations and for hybrid distributed/shared-memory supercomputers, but any computer system with a Fortran (2008+) compiler is suited
Operating system(s) for which the program has been designed: designed for POSIX architecture and tested on GNU/Linux one
RAM required to execute with typical data: bytes: $[1MB, 1GB] \times core$, simulation-dependent
Has the code been vectorised or parallelized?: the library is not aware of the parallel back-end, it providing a high-level models, but the provided tests suite shows parallel usage by means of MPI library and OpenMP paradigm
Number of processors used: tested up to 256
Supplementary material:
Keywords: ODE, PDE, OOP, ACP, Fortran
CPC Library Classification: 4.3 Differential Equations, 4.10 Interpolation, 12 Gases and Fluids
External routines/libraries used:
CPC Program Library subprograms used:
Nature of problem:
Numerical integration of (general) Ordinary Differential Equations system
Solution method:
Restrictions:
Unusual features:
Additional comments:
Running time:
References:

1. Introduction

1.1. Background

Initial Value Problem (IVP, or Cauchy problem) constitutes a class of mathematical models of paramount relevance, it being applied to the modelling of a wide range of problems. An IVP is an Ordinary Differential Equation (ODE, or system of equations, ODEs) coupled with specified initial values of the unknown state variables, the solution of which are searched at a given time after the initial time considered.

The prototype of IVP can be expressed as:

$$\begin{aligned} U_t &= R(t, U) \\ U_0 &= U(t_0) \end{aligned} \tag{1}$$

*Corresponding author
Email addresses: stefano.zaghi@cnr.it (Zaghi S.), milan@orca.rsmas.miami.edu (Curcic M.), damian@sourceryinstitute.org (Rouson D.), izaak@izaakbeekman.com (Beekman I.), giacomo.rossi@uniroma1.it (Rossi G.)
[1]Ph. D., Aerospace Engineer, Research Scientist, Dept. of Computational Hydrodynamics at CNR-INSEAN.
[2]Ph.D. Meteorology and Physical Oceanography, Research Scientist, Dept. of Ocean Sciences Rosenstiel School of Marine and Atmospheric Science at University of Miami
[3]Ph.D. Mechanical Engineering, Founder and President Sourcery Institute and Sourcery, Inc.
[4]Graduate Research Assistant, Princeton/UMD CCROCCO LAB
[5]Ph. D., Aerospace Engineer, Research Fellow at Mechanical and Aerospace Engineering Dept. at Sapienza, University of Rome

where $U(t)$ is the vector of state variables being a function of the time-like independent variable t, $U_t = \frac{dU}{dt} = R(t, U)$ is the (vectorial) residuals function and $U(t_0)$ is the (vectorial) initial conditions, namely the state variables function evaluated at the initial time t_0. In general, the residuals function R is a function of the state variable U through which it is a function of time, but it can be also a direct function of time, thus in general $R = R(t, U(t))$ holds.

The problem prototype 1 is ubiquitous in the mathematical modelling of physical problems: essentially whenever an *evolutionary* phenomenon is considered the prevision (simulation) of the future solutions involves the solution of an IVP. As a matter of facts, many physical problems (fluid dynamics, chemistry, biology, evolutionary-anthropology, ...) are described by means of an IVP.

It is worth noting that the state vector variables U and its corresponding residuals function $U_t = \frac{dU}{dt} = R(t, U)$ are *problem dependent*: the number and the meaning of the state variables as well as the equations governing their evolution (that are embedded into the residuals function) are different for the Navier-Stokes conservation laws with respect the Burgers one, as an example. Nevertheless, the *solvers* used for the prediction of the Navier-Stokes equations evolution (hereafter the *solution*) are the same that are used for Burgers equations time-integration. As a consequence, the solution of the IVP model prototype can be generalized, allowing the application of the same solver to many different problems, thus eliminating the necessity to re-implement the same solver for each different problem.

FOODIE library is designed for solving the generalized IVP 1, it being completely unaware of the actual problem's definition. FOODIE library provides a high-level, well-documented, simple Application Program Interface (API) for many well-known ODE integration solvers, its aims being twofold:

- provide a robust set of ODE solvers ready to be applied to a wide range of different problems;
- provide a simple framework for the rapid development of new ODE solvers.

Add citations to IVP, Cauchy, ODE references.

1.2. Related Works

To be written.

1.3. Motivations and Aims

FOODIE development arises from the following specifications:

- be written in modern Fortran (standard 2008 or newer);
- be abstract, thus it is written by means of Object Oriented Programming (OOP) paradigm;
- be well documented;
- be Tests Driven Developed (TDD);
- be collaboratively developed;
- be free.

Fortran (Formula Translator) programming language is the *de facto* standard into computer science field: it strongly facilitates the effective and efficient translation of (even complex) mathematical and numerical models into an operative software without compromise on computations speed and accuracy. Moreover, its simple syntax is suitable for scientific researchers that are interested (and skilled) in the physical aspects of the numerical computations rather than in the information technology aspects. Consequently, we develop FOODIE using Fortran language: FOODIE is written by research scientists for research scientists.

One key-point of the FOODIE development is the *problem abstraction*: the problem solved must be the IVP 1 rather than any of its real, concrete applications, e.g. the Lorentz system. Consequently, we must rely on a *generic* implementation of the solvers. To this aim, OOP is very useful: it allows to express IVP 1 in a very concise and clear formulation that is really generic. In particular, our implementation is based on the *Abstract Calculus Pattern* (ACP) concept.

The Abstract Calculus Pattern. The abstract calculus pattern provides a simple solution for the connection between the very high-level expression of IVP 1 and the eventual concrete (low-level) implementation of the ODE problem being solved. ACP essentially constitutes a *contract* based on an Abstract Data Type (ADT): we specify an ADT supporting a certain set of mathematical operators (differential and integral ones) and implement FOODIE solvers only on the basis of this ADT. FOODIE clients must formulate the ODE problem under integration defining their own concrete extensions of our ADT (implementing all the deferred operators). Such an approach defines the abstract calculus pattern: FOODIE solvers are aware of only the ADT, while FOODIE clients extend the ADT for defining the concrete ODE problem.

Is is worth noting that this ACP emancipates the solvers implementations from any low-level problem-dependent details: the ODE solvers developed with this pattern are extremely concise, clear, maintainable and less errors-prone with respect a low-level (non abstract) pattern. Moreover, the FOODIE clients can use solvers being extremely robust: as a matter of facts, FOODIE solvers are expressed in a formulation very close to the mathematical one and are tested on an extremely varying family of problems. As shown in the following, such a great flexibility does not compromise the computational efficiency: ACP introduces an overhead that is, however, very contained.

The present paper is organized as following: in section 2 a brief description of the mathematical and numerical methods currently implemented into FOODIE is presented; in section 3 a detailed discussion on the implementation specifications is provided by means of an analytical code-listings review; in section 4 a verification analysis on the results of FOODIE applications is presented; section 5 provides an analysis of FOODIE performances under parallel frameworks scenario like the OpenMP and MPI paradigms; finally, in section 6 concluding remarks and perspectives are depicted.

Add citations to Fortran standards, OOP, TDD, free software, ACP, ADT, Rouson's book references.

2. Mathematical and Numerical Models

In many (most) circumstances, the solution of equation 1 cannot be computed in a closed, exact form (even if it exists and is unique) due to the complexity and nature of the residuals functions, that is often non linear. Consequently, the problem is often solved relying on a numerical approach: the solution of system 1 at a time t^n, namely $U(t^n)$, is approximated by a subsequent time-marching approximations $U_0 = u_0 \rightarrow u_1 \rightarrow u_2 \rightarrow ... \rightarrow u_N \approx U(t^n)$ where the relation $u_i \rightarrow u_{i+1}$ implies a *stepping, numerical integration* from the time t^i to time t^{i+1} and N is the total number of numerical time steps necessary to evolve the initial conditions toward the searched solution $U(t^n)$. To this aim, many numerical schemes have been devised. Notably, the numerical schemes of practical usefulness must posses some necessary proprieties such as *consistency* and *stability* to ensure that the numerical approximation *converges* to the exact solution as the numerical time step tends to zero. A detailed discussion of these details is out the scope of the present work and is omitted. Here, we briefly recall some classifications necessary to introduce the schemes implemented into the FOODIE library.

A non comprehensive classification of the most widely used schemes could distinguish between *multi-step* versus *one-step* schemes and between *explicit* versus *implicit* schemes.

Essentially, the multi-step schemes have been developed to obtain an accurate approximation of the subsequent numerical steps using the informations contained into the previously computed steps, thus this approach relates the next step approximation to a set of the previously computed steps. On the contrary, a one-step scheme evolves the solution toward the next step using only the information coming from the current time approximation. In the framework of one-step schemes family an equivalent accurate approximation can be obtained by means of a multi-stage approach as the one due to Runge and Kutta. FOODIE provides schemes belonging to both these families.

The other ODE solvers classification concerns with explicit or implicit nature of the schemes employed. Briefly, an explicit scheme computes the next step approximation using the previously computed steps at most to the current time, whereas an implicit scheme uses also the next step approximation (that is the unknown), thus it requires extra computations. The implicit approach is of practical use for *stiff* systems where the usage of explicit schemes could require an extremely small time step to evolve in a *stable* way the solution. Mixing together explicit and implicit schemes it is possible to build families of *predictor-corrector* methods: using an explicit scheme to predict a guess for the next step approximation it is possible to use an implicit method for correcting this guess. FOODIE provides explicit, implicit and predictor-correct solvers.

FOODIE currently implements the following ODE schemes:

- explicit schemes:
  - forward Euler: 1^{st} order of accurate;
  - Adams-Bashforth: multi-step from 1^{st} to 16^{th} order of accuracy;
  - Leapfrog:
    - unfiltered leapfrog, 2^{nd} order accurate, mostly unstable;
    - Robert-Asselin filtered leapfrog, 1^{st} order accurate;
    - Robert-Asselin-Williams filtered leapfrog, 3^{rd} order accurate;
  - Strong Stability Preserving (SSP) Linear Multistep Methods:
    - 3 steps, 2^{nd} order accurate;
    - 4 steps, 3^{rd} order accurate;
    - 5 steps, 3^{rd} order accurate;
  - Strong Stability Preserving (SSP) Linear Multistep Methods with variable step-size:
    - from 2 to 3 steps with 2^{nd} order of accuracy;
    - from 3 to 5 steps with 3^{rd} order of accuracy;
  - Strong Stability Preserving (SSP) Linear Multistep Runge-Kutta Methods:
    - 2 steps, 2 stages, 3^{rd} order accurate;
    - 3 steps, 2 stages, 3^{rd} order accurate;

        * 4 steps, 5 stages, 8^{th} order accurate;
    - Runge-Kutta schemes:
        * Linear Strong Stability Preserving (SSP) RK of any order:
            · generic s-stages of order $(s-1)^{th}$;
            · generic s-stages of order s^{th};
        * Strong Stability Preserving (SSP) Low Storage RK:
            · 5, 6, 7, 12-14 stages, 4^{th} order accurate, 2 registers;
        * Strong Stability Preserving (SSP) RK:
            · 2 stages, 2^{nd} order accurate;
            · 3 stages, 3^{rd} order accurate;
            · 5 stages, 4^{th} order accurate;
        * embedded (adaptive) RK:
            · Heun-Euler, 2 stages, 2^{nd} order accurate;
            · Runge-Kutta-Cash-Karp, 6 stages, 5^{th} order accurate;
            · Prince-Dormand, 7 stages, 4^{th} order accurate;
            · Calvo, 9 stages, 6^{th} order accurate;
            · Feagin, 17 stages, 10^{th} order accurate;

- implicit schemes:
    - Adams-Moulton: multi-step from 1^{st} to 16^{th} order of accuracy;
    - Backward Differentiation Formula: multi-step from 2^{nd} to 6^{th} order of accuracy;
- predictor-corrector schemes:
    - Adams-Bashforth-Moulton: multi-step from 1^{st} to 16^{th} order of accuracy;

Add citations to all solvers reference papers.

In the subsequent sections each of the above mentioned scheme is briefly described.

It is worth noting that for multi-step methods like Adams-Bashforth, Adams-Moulton, Linear Multistep Methods, etc... classes the solvers are not *self-starting*: the values of $U\left(t^1\right)$, $U\left(t^2\right)$, ..., $U\left(t^{N_s-1}\right)$ must be provided, N_s being the number of previous steps used. To this aim, a lower order multi-step scheme or an equivalent order one-step multi-stage scheme can be used.

2.1. Explicit forward Euler scheme

The explicit forward Euler scheme for ODE integration is probably the simplest solver ever devised. Considering the system 1, the solution (approximation) of the state vector U at the time $t^{n+1} = t^n + \Delta t$ assuming to known the solution at time t^n is:

$$U\left(t^{n+1}\right) = U\left(t^n\right) + \Delta t \cdot R\left(t^n, U\left(t^n\right)\right) \tag{2}$$

where the solution at the new time step is computed by means of only the current time solution, thus this is an explicit scheme. The solution is an approximation of 1^{st} order, the local truncation error being $O(Dt^2)$. As well known, this scheme has an absolute (linear) stability locus equals to $|1 + \Delta t\lambda| \leq 1$ where λ contains the eigenvalues of the linear (or linearized) Jacobian matrix of the system.

This scheme is Total Variation Diminishing (TVD), thus satisfies the maximum principle or the equivalent positivity preserving property. In different fields this property is also indicated as Strong Stability Preserving.

Add citations.

2.2. Explicit Adams-Bashforth class

Adams-Bashforth methods belong to the more general (linear) explicit multi-step family of schemes. This kind of schemes has been designed to achieve a more accurate solution than the 1^{st} Euler scheme using the information coming from the solutions already computed at previous time steps. Typically only one new residuals function R evaluation is required at each time step, whereas Runge-Kutta schemes require many of them.

In general, the Adams-Bashforth schemes provided by FOODIE library are written by means of the following algorithm (for only explicit schemes):

$$U\left(t^{n+N_s}\right) = U\left(t^{n+N_s-1}\right) + \Delta t \sum_{s=1}^{n+N_s} b_s \cdot R\left[t^{n+s-1}, U\left(t^{n+s-1}\right)\right] \tag{3}$$

where N_s is the number of time steps considered and b_s are the linear coefficients selected.

Currently FOODIE provides schemes having 2^{nd} to 16^{th} formal order of accuracy. The b_s coefficients are reported in table A.12.

It is worth noting that FODDIE also provides a one-step AB solver that reverts back to the explicit forward Euler scheme: it can be used, for example, into a Recursive Order Reduction (ROR) framework that automatically checks some properties of the solution and, in case, reduces the order of the Runge-Kutta solver until those properties are obtained.

Add citations.

2.3. Leapfrog solver

Leapfrog scheme belongs to the multi-step family, it being formally a centered second order approximation in time. The leapfrog method (in its original formulation) is mostly unstable, however it is well suited for periodic-oscillatory problems providing a null error on the amplitude value and a formal second order error on the phase one, under the satisfaction of the time-step size stable limit. Commonly, the leapfrog methods are said to provide a $2\Delta t$ computational mode that can generate unphysical, unstable solutions. As consequence, the original leapfrog scheme is generally *filtered* in order to suppress these computational modes.

The unfiltered leapfrog scheme provided by FOODIE is:

$$U\left(t^{n+2}\right) = U\left(t^{n}\right) + 2\Delta t \cdot R\left[t^{n+1}, U\left(t^{n+1}\right)\right] \tag{4}$$

FOODIE provides, in a *seamless* API, also filtered leapfrog schemes. A widely used filter is due to Robert and Asselin, that suppress the computational modes at the cost of accuracy reduction resulting into a 1^{st} order error in amplitude value. A more accurate filter, able to provide a 3^{rd} order error on amplitude, is a modification of the Robert-Asselin filter due to Williams known as Robert-Asselin-Williams (RAW) filter, that filters the approximation of $U\left(t^{n+1}\right)$ and $U\left(t^{n+2}\right)$ by the following scalar coefficient:

$$\begin{gathered} U\left(t^{n+1}\right) = U\left(t^{n+1}\right) + \Delta * \alpha \\ U\left(t^{n+2}\right) = U\left(t^{n+2}\right) + \Delta * (\alpha - 1) \\ where \\ \Delta = \tfrac{\nu}{2}(U^{n} - 2U^{n+1} + U^{n+2}) \end{gathered} \tag{5}$$

The filter coefficients should be taken as $\nu \in (0, 1]$ and $\alpha \in (0.5, 1]$. If $\alpha = 0.5$ the filters of time t^{n+1} and t^{n+2} have the same amplitude and opposite sign thus allowing to the optimal 3^{rd} order error on amplitude. The default values of the FOODIE provided scheme are $\nu = 0.01$ $\alpha = 0.53$, but they can be customized at runtime.

Add citations.

2.4. Explicit SSP Linear Mutistep Methods class

Explicit SSP Linear Mutistep Methods belong to multi-step schemes like the Adams-Bashforth ones, but they ensure the Strong Stability Preserving property.

The algorithm implemented into FOODIE is:

$$U^{n+N_s} = \sum_{s=1}^{N_s} \left[a_s U^{n+s-1} + \Delta t b_s \cdot R(t^{n+s-1}, U^{n+s-1}) \right] \tag{6}$$

where N_s is the number of time steps considered and $a_s, \quad b_s$ are the linear coefficients selected.

Currently FOODIE provides schemes having 2^{nd} to 3^{th} formal order of accuracy. The $a_s, \quad b_s$ coefficients are reported in table **??**.

Add citations.

2.5. Explicit SSP Linear Mutistep Methods with variable step-size class

Explicit SSP Linear Mutistep Methods with variable step-size belong to multi-step schemes like the Adams-Bashforth ones, but they ensure the Strong Stability Preserving property and they allow the time step-size to vary, as it happens for multi-stage scheme.

Currently FOODIE provides schemes having 2^{nd} to 3^{th} formal order of accuracy. The 2^{nd} order algorithm implemented into FOODIE is:

$$U^{n+N_s} = \frac{1}{\Omega^2_{N_s-1}} U^n + \frac{\Omega^2_{N_s-1} - 1}{\Omega^2_{N_s-1}} U^{n+N_s-1} + \frac{\Omega_{N_s-1} + 1}{\Omega_{N_s-1}} \Delta t^{n+N_s} R(U^{n+N_s-1}) \tag{7}$$

while the 3^{rd} order algorithm is:

$$U^{n+N_s} = \frac{3\Omega_{N_s-1} + 2}{\Omega^3_{N_s-1}} U^n + \frac{(\Omega_{N_s-1} + 1)^2(\Omega_{N_s-1} - 2)}{\Omega^3_{N_s-1}} U^{n+N_s-1} + \frac{\Omega_{N_s-1} + 1}{\Omega^2_{N_s-1}} \Delta t^{n+N_s} R(U^n) + \frac{(\Omega_{N_s-1} + 1)^2}{\Omega^2_{N_s-1}} \Delta t^{n+N_s} R(U^{n+N_s-1}) \tag{8}$$

The coefficients are computed by recursive formula:

$$\Omega_s = \sum_{i=1}^{s} \omega_i \quad 1 \leq s \leq N_s$$

$$\omega_i = \frac{\Delta t^{n+s}}{\Delta t^{n+N_s}}$$

where N_s is the number of time steps considered.

Add citations.

2.6. Explicit SSP Linear Mutistep Methods Runge-Kutta class

Because SSP Runge-Kutta has an order barrier, see [**?**], namely they can be at most of 4^{th} order accurate, multi-step/multi-stage schemes have been devised. Such a class of schemes represent a good compromise between memory efficiency and accuracy. The schemes are not self-starting, being a multi-step class, but the same multi-step nature allows to overcome the order barrier of SSP multi-stage schemes.

The algorithm implemented into FOODIE is:

$$\begin{gathered} y^n_1 = u^n \\ y^n_i = \sum_{l=1}^{k} d_{il} u^{n-k+l} + \Delta t \sum_{l=1}^{k-1} \hat{a}_{il} F(u^{n-k+l}) + \Delta t \sum_{j=1}^{i-1} a_{ij} F(y^n_j) \quad 2 \leq i \leq N_s \\ u^{n+1} = \sum_{l=1}^{k} \theta_l u^{n-k+l} + \Delta t \sum_{l=1}^{k-1} \hat{b}_l F(u^{n-k+l}) + \Delta t \sum_{j=1}^{N_s} b_j F(y^n_j). \end{gathered} \tag{9}$$

where N_s is the number of time steps considered.

Add citations.

2.7. Explicit SSP Runge-Kutta class

Runge-Kutta methods belong to the more general multi-stage family of schemes. This kind of schemes has been designed to achieve a more accurate solution than the 1^{st} Euler scheme, but without increasing the number of time steps used, as it is done with the multi-step schemes. Essentially, the high order of accuracy is obtained by means of *intermediate values* (the stages) of the solution and its derivative are generated and used within a single time step. This commonly implies the allocation of some auxiliary memory registers for storing the intermediate stages.

Notably, the multi-stage schemes class has the attractive property to be *self-starting*: the high order accurate solution can be obtained directly from the previous one, without the necessity to compute *before* a certain number of previous steps, as it happens for the multi-step schemes. Moreover, one-step multi-stage methods are suitable for adaptively-varying time-step size (that is also possible for multi-step schemes, but at a cost of more complexity) and for discontinuous solutions, namely discontinued solutions happening at a certain time $t*$ (that in a multi-step framework can involve an overall accuracy degradation).

In general, the SSP Runge-Kutta schemes provided by FOODIE library are written by means of the following algorithm:

$$U^{n+1} = U^n + \Delta t \cdot \sum_{s=1}^{N_s} \beta_s K_s \tag{10}$$

where N_s is the number of Runge-Kutta stages used and K_s is the s^{th} stage defined as:

$$K_s = R\left(t^n + \gamma_s \Delta t, U^n + \Delta t \sum_{l=1}^{s-1} \alpha_{s,l} K_l\right) \tag{11}$$

It is worth noting that the equations 10 and 11 contain also implicit schemes. A scheme belonging to this family is operative once the coefficients α, β, γ are provided. We represent these coefficients using the Butcher's table, that for an explicit scheme where $\gamma_1 = \alpha_{1,*} = \alpha_{i,i} = 0$ has the form reported in table 1.

Table 1: Butcher's table for explicit Runge-Kutta schemes

γ_2	$\alpha_{2,1}$				
γ_3	$\alpha_{3,1}$	$\alpha_{3,2}$			
$\vdots$	$\vdots$		$\ddots$		
γ_{N_s}	$\alpha_{N_s,1}$	$\alpha_{N_s,2}$	$\cdots$	α_{N_s,N_s-1}	
	β_1	β_2	$\cdots$	β_{N_s-1}	β_{N_s}

The equations 10 and 11 show that Runge-Kutta methods do not require any additional differentiations of the ODE system for achieving high order accuracy, rather they require additional evaluations of the residuals function R.

The nature of the scheme and the properties of the solutions obtained depend on the number of stages and on the value of the coefficients selected. Currently, FOODIE provides many Runge-Kutta schemes having Strong Stability Preserving propriety (thus they being suitable for ODE systems involving rapidly changing non linear dynamics) the Butcher's coefficients of which are reported in tables A.13, A.14, A.15.

The absolute stability locus depends on the coefficients selected, however, as a general principle, we can assume that greater is the stages number and wider is the stability locus on equal accuracy orders. Currently, FOODIE provides 2^{nd} to 4^{th} order accurate schemes. As demonstrated in [**?**], SSP RK are at most 4^{th} order accurate, namely the SSP property introduces an order barrier for single step methods. To increase the order it is necessary to adopt multi-step schemes: the hybrid multi-step RK schemes are devised to this aim.

Similarly to the Adams-Bashforth class, the Runge-Kutta class also provides a fail-safe one-stage solver reverting back to the explicit forward Euler solver, that is useful for ROR-like frameworks.

Add citations.

2.8. Explicit low storage Runge-Kutta class

As aforementioned, standard Runge-Kutta schemes have the drawback to require N_S auxiliary memory registers to store the necessary stages data. In order to make an efficient use of the available limited computer memory, the class of low storage Runge-Kutta scheme was devised. Essentially, the standard Runge-Kutta class (under some specific conditions) can be reformulated allowing a more efficient memory management. Currently FOODIE provides a class of *2N registers* storage Runge-Kutta schemes, meaning that the storage of all stages requires only 2 registers of memory with a *word* length N (namely the length of the state vector) in contrast to the standard formulation where N_s registers of the same length N are required. This is a dramatic improvement of memory efficiency especially for schemes using a high number of stages ($N_s \geq 4$) where the memory necessary is an half with respect the original formulation. Unfortunately, not all standard Runge-Kutta schemes can be reformulated as a low storage one.

Following the Williamson's approach the standard coefficients are reformulated to the coefficients vectors A, B and C and the Runge-Kutta algorithm becomes:

$$\left.\begin{array}{c} K_1 = U\left(t^n\right) \\ K_2 = 0 \\ K_2 = A_s K_2 + \Delta t \cdot R\left(t^n + C_s \Delta t, K_1\right) \\ K_1 = K_1 + B_s K_2 \\ U\left(t^{n+1}\right) = K_1 \end{array}\right\} s = 1, 2, ...N_s \tag{12}$$

Currently FOODIE provides 5, 6, 7, 12, 13 and 14 stages, all 4^{th} order, 2N registers explicit schemes, the coefficients of which are listed in table A.16.

Similarly to the SSP Runge-Kutta class, the low storage class also provides a fail-safe one-stage solver reverting back to the explicit forward Euler solver, that is useful for ROR-like frameworks.

Add citations.

2.9. Implicit Adams-Moulton class

Adams-Moulton methods belong to the more general (linear) implicit multi-step family of schemes. This kind of schemes has been designed to achieve a more accurate solution than the 1^{st} Euler scheme using the information coming from the solutions already computed at previous time steps. Typically only one new residuals function R evaluation is required at each time step, whereas Runge-Kutta schemes require many of them.

In general, the Adams-Moulton schemes provided by FOODIE library are written by means of the following algorithm (for only implicit schemes):

$$U\left(t^{n+N_s}\right) = U\left(t^{n+N_s-1}\right) + \Delta t \sum_{s=0}^{n+N_s-1} b_s \cdot R\left[t^{n+s}, U\left(t^{n+s}\right)\right] + b_{N_s} \cdot R\left[t^{n+N_s}, U\left(t^{n+N_s}\right)\right] \tag{13}$$

where N_s is the number of time steps considered and b_s are the linear coefficients selected.

Currently FOODIE provides schemes having 2^{nd} to 16^{th} formal order of accuracy. The b_s coefficients are reported in table A.17.

Similarly to the Runge-Kutta and Adams-Bashforth classes, the Adams-Moulton class also provides a fail-safe zero-step solver reverting back to the implicit backward Euler solver, that is useful for ROR-like frameworks.

Add citations.

2.10. Predictor-corrector Adams-Bashforth-Moulton class

Adams-Bashforth-Moulton methods belong to the more general (linear) predictor-corrector multi-step family of schemes. This kind of schemes has been designed to achieve a more accurate solution than the 1^{st} Euler scheme using the information coming from the solutions already computed at previous time steps. Typically only one new residuals function R evaluation is required at each time step, whereas Runge-Kutta schemes require many of them.

In general, the Adams-Bashforth-Moulton schemes provided by FOODIE library are written by means of the following algorithm:

$$\begin{aligned} U\left(t^{n+N_s^p}\right)_p &= U\left(t^{n+N_s^p-1}\right) + \Delta t \sum_{s=1}^{n+N_s^p} b_s^p \cdot R\left[t^{n+s-1}, U\left(t^{n+s-1}\right)\right] \\ U\left(t^{n+N_s^c}\right)_c &= U\left(t^{n+N_s^c-1}\right) + \Delta t \sum_{s=0}^{n+N_s^c-1} b_s^c \cdot R\left[t^{n+s}, U\left(t^{n+s}\right)\right] + b_{N_s^c}^c \cdot R\left[t^{n+N_s^p}, U\left(t^{n+N_s^p}\right)_p\right] \end{aligned} \tag{14}$$

where $N_s^{p,c}$ is the number of time steps considered for the Adams-Bashforth predictor/Adams-Moulton corrector (respectively) and $b_s^{p,c}$ are the corresponding linear coefficients selected. Essentially, the Adams-Bashforth prediction $U\left(t^{n+N_s^p}\right)_p$ is corrected by means of the Adams-Moulton correction resulting in $U\left(t^{n+N_s^c}\right)_c$. In order to preserve the formal order of accuracy the relation $N_s^p = N_s^c + 1$ always holds.

Currently FOODIE provides schemes having 2^{nd} to 16^{th} formal order of accuracy. The $b_s^{p,c}$ coefficients are those reported in tables A.12 and A.17.

Add citations.

3. Application Program Interface

In this section we review the FOODIE API providing a detailed discussion of the implementation choices.

As aforementioned, the programming language used is the Fortran 2008 (or newer) standard, that is a minor revision of the previous Fortran 2003 standard. Such a new Fortran idioms provide (among other useful features) an almost complete support for OOP, in particular for Abstract Data Type (ADT) concept. Fortran 2003 has introduced the *abstract derived type*: it is a derived type suitable to serve as *contract* for concrete type-extensions that has not any actual implementations, rather it provides a well-defined set of methods (in Fortran jargon they are *type bound procedures*) interfaces, that in Fortran nomenclature are called *deferred* procedures. Using such an abstract definition, we can implement algorithms operating on only this abstract type *and on all its concrete extensions*. This is the key feature of FOODIE library: all the above described ODEs solvers are implemented on the knowledge of *only one abstract type*, allowing an implementation-style based on a very high-level syntax. In the meanwhile, client codes must implement their own IVPs extending only one simple abstract type with minimal requirements.

In the subsection 3.1 a review of the FOODIE main ADT, the *integrand_object* type, is provided, while subsections 3.2, 3.8, 3.9, 3.3 and 3.4 cover the API of the currently implemented solvers.

It is worth noting that all FOODIE public *entities* (ADT and solvers) must be accessed by the FOODIE module, see listing 1 for an example on how to access to all public FOODIE entities.

```
use foodie, only : integrand_object,                    &
                   integrator_object,                   &
                   integrator_adams_bashforth,          &
                   integrator_adams_bashforth_moulton,  &
                   integrator_adams_moulton,            &
                   integrator_back_df,                  &
                   integrator_euler_explicit,           &
                   integrator_leapfrog,                 &
                   integrator_lmm_ssp,                  &
                   integrator_lmm_ssp_vss,              &
                   integrator_ms_runge_kutta_ssp,       &
                   integrator_runge_kutta_emd,          &
                   integrator_runge_kutta_ls,           &
                   integrator_runge_kutta_lssp,         &
                   integrator_runge_kutta_ssp
! or simply
use foodie
```

Listing 1: usage example importing all public entities of FOODIE main module

3.1. Main FOODIE Abstract Data Type: the integrand type

The implemented ACP is based on one main ADT, the *integrand_object* type, the definition of which is shown in listing 2.

```
type, abstract :: integrand_object
  !< Abstract type for building FOODIE ODE integrators.
  contains
    ! public deferred procedures that concrete integrand-field must implement
    procedure(time_derivative), pass(self), deferred :: t
    ! operators
    procedure(local_error_operator), pass(lhs), deferred :: local_error
    generic :: operator(.lterror.) => local_error ! Estimate local truncation error.
    procedure(symmetric_operator), pass(lhs), deferred :: integrand_add_integrand
    procedure(integrand_op_real),  pass(lhs), deferred :: integrand_add_real
    procedure(real_op_integrand),  pass(rhs), deferred :: real_add_integrand
    generic :: operator(+) => integrand_add_integrand, &
                              integrand_add_real,      &
                              real_add_integrand
    procedure(symmetric_operator),       pass(lhs), deferred :: integrand_multiply_integrand
    procedure(integrand_op_real),        pass(lhs), deferred :: integrand_multiply_real
    procedure(real_op_integrand),        pass(rhs), deferred :: real_multiply_integrand
    procedure(integrand_op_real_scalar), pass(lhs), deferred :: integrand_multiply_real_scalar
```

```
    procedure(real_scalar_op_integrand), pass(rhs), deferred :: real_scalar_multiply_integrand
    generic :: operator(*) => integrand_multiply_integrand,   &
                              integrand_multiply_real,        &
                              real_multiply_integrand,        &
                              integrand_multiply_real_scalar, &
                              real_scalar_multiply_integrand
    procedure(symmetric_operator), pass(lhs), deferred :: integrand_sub_integrand
    procedure(integrand_op_real),  pass(lhs), deferred :: integrand_sub_real
    procedure(real_op_integrand),  pass(rhs), deferred :: real_sub_integrand
    generic :: operator(-) => integrand_sub_integrand, &
                              integrand_sub_real,      &
                              real_sub_integrand
    procedure(assignment_integrand), pass(lhs), deferred :: assign_integrand
    procedure(assignment_real),      pass(lhs), deferred :: assign_real
    generic :: assignment(=) => assign_integrand, assign_real
endtype integrand_object
```

Listing 2: integrand type definition

The *integrand_object* type does not implement any actual integrand field, it being and abstract type. It only specifies which deferred procedures are necessary for implementing an actual concrete integrand type that can use a FOODIE solver.

As shown in listing 2, the number of the deferred type bound procedures that clients must implement into their own concrete extension of the *integrand* ADT is limited: essentially, there are 1 ODEs-specific procedure and some operators definition constituted by symmetric operators between 2 integrand objects, asymmetric operators between integrand and real numbers (and viceversa) and an assignment statement for the creation of new integrand objects. These procedures are analyzed in the following paragraphs.

It is worth noting that for keeping the computational efficiency high a compromise has been done in the TBPs API design: in order to avoid excessive overhead due to polymorphic entities allocation[6] and to allows *pure*[7] operators, all the math operators are designed to return a real array rather than an allocatable class of *integrand_object*. The *integrand_object* is then *restored* by the assignment operator that updates an instance of *integrand_object* by the real array resulting from the math operators. This is a clear restriction to the ADT abstraction level: it imposes a *pattern* to the programmer that could be avoided, but, in the meanwhile, it greatly improves FOODIE performance retaining a reasonable level of abstraction.

3.1.1. Time derivative procedure, the residuals function

The abstract interface of the time derivative procedure *t* is shown in listing 3.

```
function time_derivative(self, t) result(dState_dt)
import :: integrand_object, R_P
class(integrand_object), intent(in) :: self          ! Integrand field.
real(R_P),     optional, intent(in) :: t             ! Time.
real(R_P), allocatable              :: dState_dt(:)  ! Result of the time derivative function of integrand field.
endfunction time_derivative
```

Listing 3: time derivative procedure interface

This procedure-function takes two arguments, the first passed as a *type bounded* argument, while the latter is optional, and it returns a real array of kind *R_P* (that is a parametric constant of FOODIE, it being customizable at compile-time). The passed dummy argument, *self*, is a polymorphic argument that could be any extensions of the *integrand_object* ADT. The optional argument *t* is the *time* at which the residuals function must be computed: it can be omitted in the case the residuals function does not depend directly on time.

Commonly, into the concrete implementation of this deferred abstract procedure clients embed the actual ODEs system being solved. As an example, for the Burgers equation, that is a Partial Differential Equations (PDE) system involving also a boundary value problem, this procedure embeds the spatial operator that convert the PDE to a

[6] Polymorphic allocation (and deallocation) can be very complex to optimize and it could easily generate memory leaks.

[7] A TBP can be defined *pure* when it do not have *side-effects*. Purity is important for enabling compiler optimizations (otherwise prevented), for allowing easy parallelization and for imposing a clear and easy-to-debug style to the programmer.

system of algebraic ODE. As a consequence, the eventual concrete implementation of this procedure can be of any level of complexity. Nevertheless, the FOODIE solvers are implemented only on the above abstract interface, thus emancipating the solvers implementation from any concrete complexity.

3.1.2. Symmetric operators procedures

The abstract interface of *symmetric* procedures is shown in listing 4.

```
pure function symmetric_operator(lhs, rhs) result(operator_result)
import :: integrand_object, R_P
class(integrand_object), intent(in) :: lhs                ! Left hand side.
class(integrand_object), intent(in) :: rhs                ! Right hand side.
real(R_P), allocatable              :: operator_result(:) ! Operator result.
endfunction symmetric_operator
```

Listing 4: symmetric operator procedure interface

This interface defines a class of procedures operating on 2 *integrand_object* arguments, namely it is used for the definition of the operators *multiplication*, *summation* and *subtraction* of integrand objects. These operators are used into the above described ODE solvers, for example see equations 2, 10, 3 or 4. The implementation details of such a procedures class are strictly dependent on the concrete extension of the integrand type.

3.1.3. Integrand/real and real/integrand operators procedures

The abstract interfaces of *Integrand/real and real/integrand operators* procedures are shown in listing 5.

```
pure function integrand_op_real(lhs, rhs) result(operator_result)
import :: integrand_object, R_P
class(integrand_object), intent(in) :: lhs                ! Left hand side.
real(R_P),               intent(in) :: rhs(1:)            ! Right hand side.
real(R_P), allocatable              :: operator_result(:) ! Operator result.
endfunction integrand_op_real

pure function real_op_integrand(lhs, rhs) result(operator_result)
!< Asymmetric type operator 'real.op.integrand'.
import :: integrand_object, R_P
class(integrand_object), intent(in) :: rhs                ! Right hand side.
real(R_P),               intent(in) :: lhs(1:)            ! Left hand side.
real(R_P), allocatable              :: operator_result(:) ! Operator result.
endfunction real_op_integrand

pure function integrand_op_real_scalar(lhs, rhs) result(operator_result)
!< Asymmetric type operator 'integrand.op.real'.
import :: integrand_object, R_P
class(integrand_object), intent(in) :: lhs                ! Left hand side.
real(R_P),               intent(in) :: rhs                ! Right hand side.
real(R_P), allocatable              :: operator_result(:) ! Operator result.
endfunction integrand_op_real_scalar

pure function real_scalar_op_integrand(lhs, rhs) result(operator_result)
!< Asymmetric type operator 'real.op.integrand'.
import :: integrand_object, R_P
real(R_P),               intent(in) :: lhs                ! Left hand side.
class(integrand_object), intent(in) :: rhs                ! Right hand side.
real(R_P), allocatable              :: operator_result(:) ! Operator result.
endfunction real_scalar_op_integrand
```

Listing 5: Integrand/real and real/integrand operators procedure interfaces

These foure interfaces are necessary in order to complete the *algebra* operating on the integrand object class, allowing the multiplication of an integrand object for a real number (scalar or array), circumstance that happens in all solvers, see equations 2, 10, 3 or 4. The implementation details of these procedures are strictly dependent on the concrete extension of the integrand type.

3.1.4. *Integrand assignment procedure*

The abstract interfaces of *integrand assignment* procedure is shown in listing 6.

```
pure subroutine assignment_integrand(lhs, rhs)
import :: integrand_object
class(integrand_object), intent(inout) :: lhs ! Left hand side.
class(integrand_object), intent(in)    :: rhs ! Right hand side.
endsubroutine assignment_integrand

pure subroutine assignment_real(lhs, rhs)
!< Symmetric assignment integrand = integrand.
import :: integrand_object, R_P
class(integrand_object), intent(inout) :: lhs      ! Left hand side.
real(R_P),               intent(in)    :: rhs(1:) ! Right hand side.
endsubroutine assignment_real
```

Listing 6: integrand assignment procedure interface

The assignment statement is necessary in order to complete the *algebra* operating on the integrand object class, allowing the assignment of an integrand object by another one or the assignment of an integrand object and a real array (arising from the result of other operators), circumstance that happens in all solvers, see equations 2, 10, 3 or 4. The implementation details of this assignment is strictly dependent on the concrete extension of the integrand type.

3.2. Explicit forward Euler solver API

The explicit forward Euler solver is exposed (by the FOODIE main module that must imported, see listing 1) as a single derived type (that is a standard convention for all FOODIE solvers) named *euler_explicit_integrator*. It provides the type bound procedure (also referred as *method*) *integrate* for integrating in time an *integrand* object, or any of its polymorphic concrete extensions. Consequently, for using such a solver it must be previously defined as an instance of the exposed FOODIE integrator type, see listing 7.

```
use FOODIE, only: euler_explicit_integrator
type(euler_explicit_integrator) :: integrator
```

Listing 7: definition of an explicit forward Euler integrator

Once an integrator of this type has been instantiated, it can be directly used without any initialization, for example see listing 8.

```
type(my_integrand) :: my_field
call integrator%integrate(U=my_field, Dt=0.1)
```

Listing 8: example of usage of an explicit forward Euler integrator

where *my_integrand* is a concrete (valid) extension of *integrand* ADT.

The complete implementation of the *integrate* method of the explicit forward Euler solver is reported in listing 9.

```
subroutine integrate(U, Dt, t)
class(integrand), intent(INOUT) :: U  !< Field to be integrated.
real(R_P),        intent(IN)    :: Dt !< Time step.
real(R_P), optional, intent(IN) :: t  !< Time.
U = U + U%t(t=t) * Dt
return
endsubroutine integrate
```

Listing 9: implementation of the *integrate* method of Euler solver

This method takes three arguments, the first argument is an integrand class, it being the integrand field that must integrated one-step-over in time, the second is the time step used and the third, that is optional, is the current time value that is passed to the residuals function for taking into account the cases where the time derivative explicitly depends on time. The time step is not automatically computed (for example inspecting the passed integrand field), thus its value must be externally computed and passed to the *integrate* method.

3.3. Explicit Adams-Bashforth class API

The explicit Adams-Bashforth class of solvers is exposed as a single derived type named *adams_bashforth_integrator*. This type provides three methods:

- *init*: initialize the integrator accordingly the possibilities offered by the class of solvers;
- *destroy*: destroy the integrator previously initialized, eventually freeing the allocated dynamic memory registers;
- *integrate*: integrate integrand field one-step-over in time;
- *update_previous*: auto update (cyclically) previous time steps solutions.

As common for FOODIE solvers, for using such a solver it must be previously defined as an instance of the exposed FOODIE integrator type, see listing 10.

```
use FOODIE, only: adams_bashforth_integrator
type(adams_bashforth_integrator) :: integrator
```

Listing 10: definition of an explicit Adams-Bashforth integrator

Once an integrator of this type has been instantiated, it must be initialized before used, for example see listing 11.

```
call integrator%init(steps=4)
```

Listing 11: example of initialization of an explicit Adams-Bashforth integrator

In the listing 11 a 4-steps solver has been initialized. As a matter of facts, from the equation 3 a solver belonging to this class is completely defined once the number of time steps adopted has been chosen. The complete definition of the *adams_bashforth_integrator* type is reported into listing 12. As shown, the linear coefficients are stored as allocatable arrays the values of which are initialized by the *init* method.

```
type :: adams_bashforth_integrator
  private
  integer(I_P)           :: steps=0 ! Number of time steps.
  real(R_P), allocatable :: b(:)    ! b coefficients.
  contains
    procedure, pass(self), public :: destroy
    procedure, pass(self), public :: init
    procedure, pass(self), public :: integrate
    procedure, pass(self), public :: update_previous
    final                         :: finalize
endtype adams_bashforth_integrator
```

Listing 12: definition of *adams_bashforth_integrator* type

After the solver has been initialized it can be used for integrating an integrand field, as shown in listing 13.

```
real                :: times(1:4)
type(my_integrand) :: my_field
type(my_integrand) :: previous(1:4)
call integrator%integrate(U=my_field, previous=previous, Dt=Dt, t=times)
```

Listing 13: example of usage of an Adams-Bashforth integrator

where *my_integrand* is a concrete (valid) extension of *integrand* ADT, *times* are the time at each 4 steps considered for the current one-step-over integration and *previous* are the memory registers where previous time steps solutions are saved.

The complete implementation of the *integrate* method of the explicit Adams-Bashforth class of solvers is reported in listing 14.

```
subroutine integrate(self, U, previous, Dt, t, autoupdate)
class(adams_bashforth_integrator), intent(IN)    :: self          ! Actual AB integrator.
class(integrand),                  intent(INOUT) :: U             ! Field to be integrated.
class(integrand),                  intent(INOUT) :: previous(1:)  ! Previous time steps solutions.
real(R_P),                         intent(IN)    :: Dt            ! Time steps.
real(R_P),                         intent(IN)    :: t(:)          ! Times.
logical, optional,                 intent(IN)    :: autoupdate    ! Autoupdate previous time steps.
logical                                          :: autoupdate_   ! autoupdate previous time steps, dummy var.
integer(I_P)                                     :: s             ! Steps counter.
autoupdate_ = .true. ; if (present(autoupdate)) autoupdate_ = autoupdate
do s=1, self%steps
  U = U + previous(s)%t(t=t(s)) * (Dt * self%b(s))
enddo
if (autoupdate_) call self%update_previous(U=U, previous=previous)
return
endsubroutine integrate
```

Listing 14: implementation of the *integrate* method of explicit Adams-Bashforth class

This method takes five arguments, the first argument is passed as bounded argument and it is the solver itself, the second is of an integrand class, it being the integrand field that must integrated one-step-over in time, the third are the previous time steps solutions, the fourth is the time step used, the fifth is an array of the time values of the steps considered for the current one-step-over integration that are passed to the residuals function for taking into account the cases where the time derivative explicitly depends on time and the sixth is a logical flag for enabling/disabling the cyclic update of previous time steps solutions. The time step is not automatically computed (for example inspecting the passed integrand field), thus its value must be externally computed and passed to the *integrate* method.

It is worth noting that the method also performs the cyclic update of the previous time steps solutions memory registers. This can be disable passing *autoupdate=.false.*: it is useful in the framework of predictor-corrector solvers.

3.4. Leapfrog solver

The explicit Leapfrog class of solvers is exposed as a single derived type named *leapfrog_integrator*. This type provides three methods:

- *init*: initialize the integrator accordingly the possibilities offered by the class of solvers;
- *integrate*: integrate integrand field one-step-over in time.

As common for FOODIE solvers, for using such a solver it must be previously defined as an instance of the exposed FOODIE integrator type, see listing 15.

```
use FOODIE, only: leapfrog_integrator
type(leapfrog_integrator) :: integrator
```

Listing 15: definition of an explicit Leapfrog integrator

Once an integrator of this type has been instantiated, it must be initialized before used, for example see listing 16.

```
! default coefficients nu=0.01, alpha=0.53
call integrator%init()
! custom coefficients
call integrator%init(nu=0.015, alpha=0.6)
```

Listing 16: example of initialization of an explicit Leapfrog integrator

The complete definition of the *leapfrog_integrator* type is reported into listing 17. As shown, the filter coefficients are initialized to zero, suitable values are initialized by the *init* method.

```
type :: leapfrog_integrator
  private
  real(R_P) :: nu=0.01_R_P     ! Robert-Asselin filter coefficient.
  real(R_P) :: alpha=0.53_R_P  ! Robert-Asselin-Williams filter coefficient.
  contains
    procedure, pass(self), public :: init
    procedure, pass(self), public :: integrate
endtype leapfrog_integrator
```

Listing 17: definition of *leapfrog_integrator* type

After the solver has been initialized it can be used for integrating an integrand field, as shown in listing 18.

```
real                :: times(1:2)
type(my_integrand)  :: filter_displacement
type(my_integrand)  :: my_field
type(my_integrand)  :: previous(1:2)
call integrator%integrate(U=my_field, previous=previous, filter=filter_displacement, Dt=Dt, &
                          t=times)
```

Listing 18: example of usage of a Leapfrog integrator

where *my_integrand* is a concrete (valid) extension of *integrand* ADT, *previous* are the memory registers where previous time steps solutions are saved, *filter_displacement* is the register necessary for computing the eventual displacement of the applied filter and *times* are the time at each 2 steps considered for the current one-step-over integration.

The complete implementation of the *integrate* method of the explicit Leapfrog class of solvers is reported in listing 19.

```
subroutine integrate(self, U, previous, Dt, t, filter)
class(leapfrog_integrator), intent(IN)    :: self           ! LF integrator.
class(integrand),           intent(INOUT) :: U              ! Field to be integrated.
class(integrand),           intent(INOUT) :: previous(1:2)  ! Previous time steps solutions.
real(R_P),                  intent(in)    :: Dt             ! Time step.
```

```
real(R_P),                    intent(IN)    :: t              ! Time.
class(integrand), optional, intent(INOUT) :: filter         ! Filter field displacement.
U = previous(1) + previous(2)%t(t=t) * (Dt * 2._R_P)
if (present(filter)) then
  filter = (previous(1) - previous(2) * 2._R_P + U) * self%nu * 0.5_R_P
  previous(2) = previous(2) + filter * self%alpha
  U = U + filter * (self%alpha - 1._R_P)
endif
previous(1) = previous(2)
previous(2) = U
return
endsubroutine integrate
```

Listing 19: implementation of the *integrate* method of explicit Leapfrog class

This method takes six arguments, the first argument is passed as bounded argument and it is the solver itself, the second is of an integrand class, it being the integrand field that must integrated one-step-over in time, the third are the previous time steps solutions, the fourth is the optional filter-displacement-register, the fifth is the time step used and the sixth is an array of the time values of the steps considered for the current one-step-over integration that are passed to the residuals function for taking into account the cases where the time derivative explicitly depends on time. The time step is not automatically computed (for example inspecting the passed integrand field), thus its value must be externally computed and passed to the *integrate* method. It is worth noting that if the filter displacement argument is not passed, the solver reverts back to the standard unfiltered Leapfrog method.

It is worth noting that the method also performs the cyclic update of the previous time steps solutions memory registers. In particular, if the filter displacement argument is passed the method performs the RAW filtering.

3.5. Explicit SSP Linear Mutistep Methods class API

To be written.

3.6. Explicit SSP Linear Mutistep Methods with variable step-size class API

To be written.

3.7. Explicit SSP Linear Mutistep Methods Runge-Kutta class API

To be written.

3.8. Explicit SSP Runge-Kutta class API

The TVD/SSP Runge-Kutta class of solvers is exposed as a single derived type named *tvd_runge_kutta_integrator*. This type provides three methods:

- *init*: initialize the integrator accordingly the possibilities offered by the class of solvers;
- *destroy*: destroy the integrator previously initialized, eventually freeing the allocated dynamic memory registers;
- *integrate*: integrate integrand field one-step-over in time.

As common for FOODIE solvers, for using such a solver it must be previously defined as an instance of the exposed FOODIE integrator type, see listing 20.

```
use FOODIE, only: tvd_runge_kutta_integrator
type(tvd_runge_kutta_integrator) :: integrator
```

Listing 20: definition of an explicit TVD/SSP Runge-Kutta integrator

Once an integrator of this type has been instantiated, it must be initialized before used, for example see listing 21.

```
call integrator%init(stages=3)
```

Listing 21: example of initialization of an explicit TVD/SSP Runge-Kutta integrator

In the listing 21 a 3-stages solver has been initialized. As a matter of facts, from the equations 10 and 11 a solver belonging to this class is completely defined once the number of stages adopted has been chosen. The complete definition of the *tvd_runge_kutta_integrator* type is reported into listing 22. As shown, the Butcher's coefficients are stored as allocatable arrays the values of which are initialized by the *init* method.

```
type :: tvd_runge_kutta_integrator
  integer(I_P)           :: stages=0   ! Number of stages.
  real(R_P), allocatable :: alph(:,:) ! alpha Butcher's coefficients.
  real(R_P), allocatable :: beta(:)    ! beta Butcher's coefficients.
  real(R_P), allocatable :: gamm(:)    ! gamma Butcher's coefficients.
  contains
    procedure, pass(self), public :: destroy
    procedure, pass(self), public :: init
    procedure, pass(self), public :: integrate
    final                         :: finalize
endtype tvd_runge_kutta_integrator
```

Listing 22: definition of *tvd_runge_kutta_integrator* type

After the solver has been initialized it can be used for integrating an integrand field, as shown in listing 23.

```
type(my_integrand) :: my_field
type(my_integrand) :: my_stages(1:3)
call integrator%integrate(U=my_field, stage=my_stage, Dt=0.1)
```

Listing 23: example of usage of a TVD/SSP Runge-Kutta integrator

where *my_integrand* is a concrete (valid) extension of *integrand* ADT. Listing 23 shows that the memory registers necessary for storing the Runge-Kutta stages must be supplied by the client code.

The complete implementation of the *integrate* method of the explicit TVD/SSP Runge-Kutta class of solvers is reported in listing 24.

```
subroutine integrate(self, U, stage, Dt, t)
class(tvd_runge_kutta_integrator), intent(IN)    :: self       ! Actual RK integrator.
class(integrand),                  intent(INOUT) :: U          ! Field to be integrated.
class(integrand),                  intent(INOUT) :: stage(1:) ! Runge-Kutta stages [1:stages].
real(R_P),                         intent(IN)    :: Dt         ! Time step.
real(R_P),                         intent(IN)    :: t          ! Time.
integer(I_P)                                     :: s          ! First stages counter.
integer(I_P)                                     :: ss         ! Second stages counter.
select type(stage)
class is(integrand)
  do s=1, self%stages
    stage(s) = U
    do ss=1, s - 1
      stage(s) = stage(s) + stage(ss) * (Dt * self%alph(s, ss))
    enddo
    stage(s) = stage(s)%t(t=t + self%gamm(s) * Dt)
  enddo
  do s=1, self%stages
    U = U +  stage(s) * (Dt * self%beta(s))
  enddo
endselect
return
endsubroutine integrate
```

Listing 24: implementation of the *integrate* method of explicit TVD/SSP Runge-Kutta class

This method takes five arguments, the first argument is passed as bounded argument and it is the solver itself, the second is of an integrand class, it being the integrand field that must integrated one-step-over in time, the third is the stages array for storing the stages computations, the fourth is the time step used and the fifth, that is optional, is the current time value that is passed to the residuals function for taking into account the cases where the time derivative explicitly depends on time. The time step is not automatically computed (for example inspecting the passed integrand field), thus its value must be externally computed and passed to the *integrate* method.

It is worth noting that the stages memory registers, namely the array *stage*, must be passed as argument because it is defined as a *not-passed* polymorphic argument, thus we are not allowed to define it as an automatic array of the *integrate* method.

3.9. Explicit low storage Runge-Kutta class API

The low storage variant of Runge-Kutta class of solvers is exposed as a single derived type named *ls_runge_kutta_integrator*. This type provides three methods:

- *init*: initialize the integrator accordingly the possibilities offered by the class of solvers;
- *destroy*: destroy the integrator previously initialized, eventually freeing the allocated dynamic memory registers;
- *integrate*: integrate integrand field one-step-over in time.

As common for FOODIE solvers, for using such a solver it must be previously defined as an instance of the exposed FOODIE integrator type, see listing 25.

```
use FOODIE, only: ls_runge_kutta_integrator
type(ls_runge_kutta_integrator) :: integrator
```

Listing 25: definition of an explicit low storage Runge-Kutta integrator

Once an integrator of this type has been instantiated, it must be initialized before used, for example see listing 26.

```
call integrator%init(stages=5)
```

Listing 26: example of initialization of an explicit low storage Runge-Kutta integrator

In the listing 26 a 5-stages solver has been initialized. As a matter of facts, from the equation 12 a solver belonging to this class is completely defined once the number of stages adopted has been chosen. The complete definition of the *ls_runge_kutta_integrator* type is reported into listing 27. As shown, the Williamson's coefficients are stored as allocatable arrays the values of which are initialized by the *init* method.

```
type :: ls_runge_kutta_integrator
  integer(I_P)           :: stages=0 ! Number of stages.
  real(R_P), allocatable :: A(:)     ! Low storage *A* coefficients.
  real(R_P), allocatable :: B(:)     ! Low storage *B* coefficients.
  real(R_P), allocatable :: C(:)     ! Low storage *C* coefficients.
  contains
    procedure, pass(self), public :: destroy
    procedure, pass(self), public :: init
    procedure, pass(self), public :: integrate
    final                         :: finalize
endtype ls_runge_kutta_integrator
```

Listing 27: definition of *ls_runge_kutta_integrator* type

After the solver has been initialized it can be used for integrating an integrand field, as shown in listing 28.

```
type(my_integrand) :: my_field
type(my_integrand) :: my_stages(1:2)
call integrator%integrate(U=my_field, stage=my_stage, Dt=0.1)
```

Listing 28: example of usage of a low storage Runge-Kutta integrator

where *my_integrand* is a concrete (valid) extension of *integrand* ADT. Listing 28 shows that the memory registers necessary for storing the Runge-Kutta stages must be supplied by the client code, as it happens of the TVD/SSP Runge-Kutta class. However, now the registers necessary is always 2, independently on the number of stages used, that in the example considered are 5.

The complete implementation of the *integrate* method of the explicit low storage Runge-Kutta class of solvers is reported in listing 29.

```
subroutine integrate(self, U, stage, Dt, t)
class(ls_runge_kutta_integrator), intent(IN)    :: self        ! Actual RK integrator.
class(integrand),                 intent(INOUT) :: U           ! Field to be integrated.
class(integrand),                 intent(INOUT) :: stage(1:2)  ! Runge-Kutta registers [1:2].
real(R_P),                        intent(IN)    :: Dt          ! Time step.
real(R_P),                        intent(IN)    :: t           ! Time.
integer(I_P)                                    :: s           ! First stages counter.
select type(stage)
class is(integrand)
  stage(1) = U
  stage(2) = U*0._R_P
  do s=1, self%stages
    stage(2) = stage(2) * self%A(s) + stage(1)%t(t=t + self%C(s) * Dt) * Dt
    stage(1) = stage(1) + stage(2) * self%B(s)
  enddo
  U = stage(1)
endselect
return
endsubroutine integrate
```

Listing 29: implementation of the *integrate* method of explicit low storage Runge-Kutta class

This method takes five arguments, the first argument is passed as bounded argument and it is the solver itself, the second is of an integrand class, it being the integrand field that must integrated one-step-over in time, the third is the stages array for storing the stages computations, the fourth is the time step used and the fifth, that is optional, is the current time value that is passed to the residuals function for taking into account the cases where the time derivative explicitly depends on time. The time step is not automatically computed (for example inspecting the passed integrand field), thus its value must be externally computed and passed to the *integrate* method.

It is worth noting that the stages memory registers, namely the array *stage*, must be passed as argument because it is defined as a *not-passed* polymorphic argument, thus we are not allowed to define it as an automatic array of the *integrate* method.

3.10. Implicit Adams-Moulton class API

The explicit Adams-Moulton class of solvers is exposed as a single derived type named *adams_moulton_integrator*. This type provides three methods:

- *init*: initialize the integrator accordingly the possibilities offered by the class of solvers;
- *destroy*: destroy the integrator previously initialized, eventually freeing the allocated dynamic memory registers;
- *integrate*: integrate integrand field one-step-over in time;
- *update_previous*: auto update (cyclically) previous time steps solutions.

As common for FOODIE solvers, for using such a solver it must be previously defined as an instance of the exposed FOODIE integrator type, see listing 30.

```
use FOODIE, only: adams_moulton_integrator
type(adams_moulton_integrator) :: integrator
```

Listing 30: definition of an implicit Adams-Moulton integrator

Once an integrator of this type has been instantiated, it must be initialized before used, for example see listing 31.

```
call integrator%init(steps=3)
```

Listing 31: example of initialization of an implicit Adams-Moulton integrator

In the listing 31 a 3-steps solver has been initialized. As a matter of facts, from the equation 13 a solver belonging to this class is completely defined once the number of time steps adopted has been chosen. The complete definition of the *adams_moulton_integrator* type is reported into listing 32. As shown, the linear coefficients are stored as allocatable arrays the values of which are initialized by the *init* method.

```
type :: adams_moulton_integrator
  private
  integer(I_P)           :: steps=-1 ! Number of time steps.
  real(R_P), allocatable :: b(:)     ! b coefficients.
  contains
    procedure, pass(self), public :: destroy
    procedure, pass(self), public :: init
    procedure, pass(self), public :: integrate
    procedure, pass(self), public :: update_previous
    final                         :: finalize
endtype adams_moulton_integrator
```

Listing 32: definition of *adams_moulton_integrator* type

After the solver has been initialized it can be used for integrating an integrand field, as shown in listing 33.

```
real                :: times(1:3)
type(my_integrand)  :: my_field
type(my_integrand)  :: previous(1:3)
call integrator%integrate(U=my_field, previous=previous, Dt=Dt, t=times)
```

Listing 33: example of usage of an Adams-Moulton integrator

where *my_integrand* is a concrete (valid) extension of *integrand* ADT, *times* are the time at each 4 steps considered for the current one-step-over integration and *previous* are the memory registers where previous time steps solutions are saved.

The complete implementation of the *integrate* method of the implicit Adams-Moulton class of solvers is reported in listing 34.

```
subroutine integrate(self, U, previous, Dt, t, autoupdate)
class(adams_bashforth_integrator), intent(IN)    :: self          ! Actual AB integrator.
class(integrand),                  intent(INOUT) :: U             ! Field to be integrated.
class(integrand),                  intent(INOUT) :: previous(1:)  ! Previous time steps solutions.
real(R_P),                         intent(IN)    :: Dt            ! Time steps.
real(R_P),                         intent(IN)    :: t(:)          ! Times.
logical, optional,                 intent(IN)    :: autoupdate    ! Autoupdate previous time steps.
logical                                          :: autoupdate_   ! autoupdate previous time steps, dummy var.
integer(I_P)                                     :: s             ! Steps counter.
autoupdate_ = .true. ; if (present(autoupdate)) autoupdate_ = autoupdate
if (autoupdate_) call self%update_previous(U=U, previous=previous)
if (self%steps>0) then
  U = previous(self%steps) + U%t(t=t(self%steps) + Dt) * (Dt * self%b(self%steps))
  do s=0, self%steps - 1
    U = U + previous(s+1)%t(t=t(s+1)) * (Dt * self%b(s))
  enddo
  if (autoupdate_) call self%update_previous(U=U, previous=previous)
else
  U = U + U%t(t=t(s+1)) * (Dt * self%b(0))
endif
return
endsubroutine integrate
```

Listing 34: implementation of the *integrate* method of explicit Adams-Moulton class

This method takes six arguments, the first argument is passed as bounded argument and it is the solver itself, the second is of an integrand class, it being the integrand field that must integrated one-step-over in time, the third are the previous time steps solutions, the fourth is the time step used, the fifth is an array of the time values of the steps considered for the current one-step-over integration that are passed to the residuals function for taking into account the cases where the time derivative explicitly depends on time and the sixth is a logical flag for enabling/disabling the cyclic update of previous time steps solutions. The time step is not automatically computed (for example inspecting the passed integrand field), thus its value must be externally computed and passed to the *integrate* method.

It is worth noting that the method also performs the cyclic update of the previous time steps solutions memory registers. This can be disable passing *autoupdate=.false.*: it is useful in the framework of predictor-corrector solvers.

3.11. Predictor-corrector Adams-Bashforth-Moulton class API

To be written.

3.12. General Remarks

Table 2 presents a comparison of the relevant parts of equations 2, 10, 11, 12, 3 and 4 with the corresponding FOODIE implementations reported in listings 9, 24, 29, 14 and 19, respectively. This comparison proves that the *integrand* ADT has allowed a very high-level implementation syntax. The Fortran implementation is almost equivalent to the rigorous mathematical formulation. This aspect directly implies that the implementation of a ODE solver into the FOODIE library is very clear, concise and less-errors-prone than an *hard-coded* implementation where the solvers must be implemented for each specific definition of the integrand type, it being not abstract.

Table 2: Comparison between rigorous mathematical formulation and FOODIE high-level implementation; the syntax "(s)" and "(ss)" imply the summation operation

Solver	Mathematical formulation	FOODIE implementation
explicit forward Euler	$U\left(t^{n+1}\right) = U\left(t^n\right) + \Delta t \cdot R\left[t^n, U\left(t^n\right)\right]$	$U = U + U\%t(t = t) * Dt$
TVD/SSP Runge-Kutta	$K_s = R\left(t^n + \gamma_s \Delta t, U^n + \Delta t \sum_{l=1}^{s-1} \alpha_{s,l} K_l\right)$ $U^{n+1} = U^n + \Delta t \cdot \sum_{s=1}^{N_s} \beta_s K_s$	$stage(s) = stage(s) + stage(ss) * (Dt * self\%alph(s, ss))$ $U = U + stage(s) * (Dt * self\%beta(s))$
low storage Runge-Kutta	$K_2 = A_s K_2 + \Delta t \cdot R\left(t^n + C_s \Delta t, K_1\right)$ $K_1 = K_1 + B_s K_2$	$stage(2) = stage(2) * self\%A(s) +$ $+stage(1)\%t(t = t + self\%C(s) * Dt) * Dt$ $stage(1) = stage(1) + stage(2) * self\%B(s)$
explicit Adams-Bashforth	$U\left(t^{n+N_s}\right) = U\left(t^{n+N_s-1}\right) +$ $+\Delta t \sum_{s=1}^{n+N_s} b_s \cdot R\left[t^{n+s-1}, U\left(t^{n+s-1}\right)\right]$	$U = U + U\%t(n = s, t = t(s)) * (Dt * self\%b(s))$
explicit Leapfrog	$U\left(t^{n+2}\right) = U\left(t^n\right) + 2\Delta t \cdot R\left[t^{n+1}, U\left(t^{n+1}\right)\right]$	$U = U\%previous\_step(n = 1) + U\%t(n = 2, t = t) * (Dt * 2.)$

4. Tests and Examples

For the assessment of FOODIE capabilities three different tests are discussed:

- Oscillation equations IVP: this test is a pure ODEs, non-stiff problem;
- Lorenz equations IVP: this test is a pure ODEs, stiff problem;
- 1D Euler equations IVP: this test is a PDEs problem;

With the first two pure ODEs tests we assess the accuracy of the FOODIE built-in solvers and demonstrate how to use the library in real applications. The last test discussion is mainly devoted to the demonstration of FOODIE application into a real, complex PDEs problem.

4.1. Oscillation equations test

Let us consider the *oscillation* problem, it being a simple, yet interesting IVP. Briefly, the oscillation problem is a prototype problem of non dissipative, oscillatory phenomena. For example, let us consider a pendulum subjected to the Coriolis accelerations without dissipation, the motion equations of which can be described by the ODE system 15.

$$\begin{aligned} U_t &= R(U) \\ U = \begin{bmatrix} x \\ y \end{bmatrix} \quad R(U) &= \begin{bmatrix} -fy \\ fx \end{bmatrix} \end{aligned} \tag{15}$$

where the frequency is chosen as $f = 10^4$. The ODE system 15 is completed by the following initial conditions:

$$\begin{aligned} x(t_0) &= 0 \\ y(t_0) &= 1 \end{aligned} \tag{16}$$

where $t_0 = 0$ is the initial time considered.

The IVP constituted by equations 15 and 16 is (apparently) simple and its exact solution is known:

$$\begin{aligned} x(t_0 + \Delta t) &= X_0 cos(f\Delta t) - y_0 sin(f\Delta t) \\ y(t_0 + \Delta t) &= X_0 sin(f\Delta t) + y_0 cos(f\Delta t) \end{aligned} \tag{17}$$

where Δt is an arbitrary time step. This problem is non-stiff meaning that the solution is constituted by only one time-scale, namely the single frequency f.

This problem is only apparently simple. As a matter of facts, in a non dissipative oscillatory problem the eventual errors in the amplitude approximation can rapidly drive the subsequent series of approximations to an unphysical solution. This is of particular relevance if the solution (that is numerically approximated) constitutes a *prediction* far from the initial conditions, that is the common case in weather forecasting.

Because the Oscillation system 15 posses a closed exact solution, the discussion on this test has twofolds aims: to assess the accuracy of the FOODIE's built-in solvers comparing the numerical solutions with the exact one and to demonstrate how it is simple to solve this prototypical problem by means of FOODIE.

4.1.1. Errors Analysis

For the analysis of the accuracy of each solver, we have integrated the Oscillation equations 15 with different, decreasing time steps in the range $[5000, 2500, 1250, 625, 320, 100]$. The error is estimated by the L2 norm of the difference between the exact (U_e) and the numerical ($U_{\Delta t}$) solutions for each time step:

$$\varepsilon(\Delta t) = \|U_e - U_{\Delta t}\|_2 = \sqrt{\sum_{s=1}^{N_s} (U_e(t_0 + s * \Delta t) - U_{\Delta t}(t_0 + s * \Delta t))^2} \tag{18}$$

where N_s is the total number of time steps performed to reach the final integration time.

Using two pairs of subsequent-decreasing time steps solution is possible to estimate the order of accuracy of the solver employed computing the *observed order* of accuracy:

$$p = \frac{log10\left(\frac{\varepsilon(\Delta t_1)}{\varepsilon(\Delta t_2)}\right)}{log10\left(\frac{\Delta t_1}{\Delta t_2}\right)} \tag{19}$$

where $\frac{\Delta t_1}{\Delta t_2} > 1$.

4.1.2. FOODIE aware implementation of an oscillation numerical solver

The IVP 15 can be easily solved by means of FOODIE library. The first block of a FOODIE aware solution consists to define an *oscillation integrand field* defining a concrete extension of the FOODIE *integrand* type. Listing 35 reports the implementation of such an integrand field that is contained into the tests suite shipped within the FOODIE library.

```
type, extends(integrand) :: oscillation
  private
  integer(I_P)                              :: dims=0    ! Space dimensions.
  real(R_P)                                 :: f=0._R_P  ! Oscillation frequency (Hz).
  real(R_P), dimension(:),   allocatable    :: U         ! Integrand (state) variables, [1:dims].
  contains
    ! auxiliary methods
    procedure, pass(self), public :: init
    procedure, pass(self), public :: output
    ! type_integrand deferred methods
    procedure, pass(self), public :: t => dOscillation_dt
    procedure, pass(lhs),  public :: integrand_multiply_integrand => &
                                     oscillation_multiply_oscillation
    procedure, pass(lhs),  public :: integrand_multiply_real => oscillation_multiply_real
    procedure, pass(rhs),  public :: real_multiply_integrand => real_multiply_oscillation
    procedure, pass(lhs),  public :: add => add_oscillation
    procedure, pass(lhs),  public :: sub => sub_oscillation
    procedure, pass(lhs),  public :: assign_integrand => oscillation_assign_oscillation
    procedure, pass(lhs),  public :: assign_real => oscillation_assign_real
endtype oscillation
```

Listing 35: implementation of the *oscillation integrand* type

The *oscillation* field extends the *integrand* ADT making it a concrete type. This derived type is very simple: it has 5 data members for storing the state vector and some auxiliary variables, and it implements all the deferred methods necessary for defining a valid concrete extension of the *integrand* ADT (plus 2 auxiliary methods that are not relevant for our discussion). The key point is here constituted by the implementation of the deferred methods: the *integrand* ADT does not impose any structure for the data members, that are consequently free to be customized by the client code. In this example the data members have a very simple, clean and concise structure:

- *dims* is the number of space dimensions; in the case of equation 15 we have $dims = 2$, however the test implementation has been kept more general parametrizing this dimension in order to easily allow future modification of the test-program itself;
- f stores the frequency of the oscillatory problem solved, that is here set to 10^4, but it can be changed at runtime in the test-program;
- U is the state vector corresponding directly to the state vector of equation 15;

As the listing 35 shows, the FOODIE implementation strictly corresponds to the mathematical formulation embracing all the relevant mathematical aspects into one derived type, a single *object*. Here we not review the implementation of all deferred methods, this being out of the scope of the present work: the interested reader can see the tests suite sources shipped within the FOODIE library. However, some of these methods are relevant for our discussion, thus they are reviewed.

dOscillation_dt, the oscillation residuals function. Probably, the most important methods for an IVP solver is the residuals function. As a matter of facts, the ODE equations are implemented into the residuals function. However, the FOODIE ADT strongly alleviates the subtle problems that could arise when the ODE solver is hard-implemented within the specific ODE equations. As a matter of facts, the *integrand* ADT specifies the precise interface the residuals function must have: if the client code implements a compliant interface, the FOODIE solvers will work as expected, reducing the possible errors location into the ODE equations, having designed the solvers on the ADT and not on the concrete type.

```
function dOscillation_dt(self, t) result(dState_dt)
class(oscillation),     intent(IN) :: self      ! Oscillation field.
real(R_P),    optional, intent(IN) :: t         ! Time.
class(integrand), allocatable      :: dState_dt ! Oscillation field time derivative.
integer(I_P)                       :: dn        ! Time level, dummy variable.
allocate(oscillation :: dState_dt)
select type(dState_dt)
class is(oscillation)
  dState_dt = self
  dState_dt%U(1) = -self%f * self%U(2)
  dState_dt%U(2) =  self%f * self%U(1)
endselect
return
endfunction dOscillation_dt
```

Listing 36: implementation of the *oscillation integrand* residuals function

Listing 36 reports the implementation of the oscillation residuals function: it is very clear and concise. Moreover, comparing this listing with the equation 15 the close correspondence between the mathematical formulation and Fortran implementation is evident.

add method, an example of oscillation symmetric operator. As a prototype of the operators overloading let us consider the *add* operator, it being a prototype of symmetric operators, the implementation of which is presented in listing 37.

```
function add_oscillation(lhs, rhs) result(opr)
class(oscillation), intent(IN) :: lhs ! Left hand side.
class(integrand),   intent(IN) :: rhs ! Right hand side.
class(integrand), allocatable  :: opr ! Operator result.
allocate(oscillation :: opr)
select type(opr)
class is(oscillation)
  opr = lhs
  select type(rhs)
  class is (oscillation)
    opr%U = lhs%U + rhs%U
  endselect
endselect
return
endfunction add_Oscillation
```

Listing 37: implementation of the *oscillation integrand* add operator

It is very simple and clear: firstly all the auxiliary data are copied into the operator result, then the state vector of the result is populated with the addiction between the state vectors of the left-hand-side and right-hand-side. This is very intuitive from the mathematical point of view and it helps to reduce implementation errors. Similar implementations are possible for all the other operators necessary to define a valid *intregrand* ADT concrete extension.

assignment of an oscillation object. The assignment overloading of the *oscillation* type is the last key-method that enforces the conciseness of the FOODIE aware implementation. Listing 38 reports the implementation of the assignment overloading. Essentially, to all the data members of the left-hand-side are assigned the values of the corresponding right-hand-side. Notably, for the assignment of the state vector and of the previous time steps solution array we take advantage of the automatic re-allocation of the left-hand-side variables when they are not allocated or allocated differently from the right-hand-side, that is a Fortran 2003 feature. In spite its simplicity, the assignment overloading is a

key-method enabling the usage of FOODIE solver: effectively, the assignment between two *integrand* ADT variables is ubiquitous into the solvers implementations, see equation 10 for example.

```
subroutine oscillation_assign_oscillation(lhs, rhs)
class(oscillation), intent(INOUT) :: lhs ! Left hand side.
class(integrand),   intent(IN)    :: rhs ! Right hand side.
select type(rhs)
class is (oscillation)
  lhs%dims = rhs%dims
  lhs%f = rhs%f
  if (allocated(rhs%U)) lhs%U = rhs%U
endselect
return
endsubroutine oscillation_assign_oscillation
```

Listing 38: implementation of the *oscillation integrand* assignment

FOODIE numerical integration. Using the above discussed *oscillation* type it is very easy to solve IVP 15 by means of FOODIE library. Listing 39 presents the numerical integration of system 15 by means of the Leapfrog RAW-filtered method. In the example, the integration is performed with 10^4 steps with a fixed $\Delta t = 10^2$ until the time $t = 10^6$ is reached. The example shows also that for starting a multi-step scheme such as the Leapfrog one a lower-oder or equivalent order one-scheme is necessary: in the example the first 2 steps are computed by means of one-step TVD/SSP Runge-Kutta 2-stages schemes. Note that the memory registers for storing the Runge-Kutta stages and the RAW filter displacement must be handled by the client code. Listing 39 demonstrates how it is simple, clear and concise to solve a IVP by FOODIE solvers. Moreover, it proves how it is simple and effective to apply different solvers in a coupled algorithm, that greatly simplify the development of new hybrid solvers for self-adaptive time step size.

```
use foodie, only: leapfrog_integrator, tvd_runge_kutta_integrator
type(leapfrog_integrator)         :: lf_integrator ! Leapfrog integrator.
type(tvd_runge_kutta_integrator) :: rk_integrator ! Runge-Kutta integrator.
type(oscillation)                 :: rk_stage(1:2) ! Runge-Kutta stages.
type(oscillation)                 :: previous(1:2) ! Previous time steps solution.
type(oscillation)                 :: oscillator    ! Oscillation field.
type(oscillation)                 :: filter        ! Filter displacement.
integer                           :: step          ! Time steps counter.
real                              :: Dt            ! Time step.
call lf_integrator%init()
call rk_integrator%init(stages=2)
call oscillator%init(initial_state=[0.0,1.0], f=10e4, steps=2)
Dt = 100.0
do step=1, 10000
  if (2>=step) then
    call rk_integrator%integrate(U=oscillator, stage=rk_stage, Dt=Dt, t=step*Dt)
    previous(step) = oscillator
  else
    call lf_integrator%integrate(U=oscillator, previous=previous, filter=filter, Dt=Dt, &
                                 t=step*Dt)
  endif
enddo
call print_results(U=oscillator)
```

Listing 39: numerical integration of the *oscillation* system by means of Leapfrog RAW-filtered method

4.1.3. Adams-Bashforth

Table 3 summarizes the Adams-Bashforth error analysis. As expected, the Adams-Bashforth 1 step solution, that reverts back to the explicit forward Euler one, is unstable for all the Δt exercised.

The expected observed orders of accuracy for the Adams-Bashforth solvers using 2, 3 and 4 time steps tend to 1.5, 2.5 and 3.5 that are in agreement with the expected formal order of 2, 3 and 4, respectively. Comparing the errors of the finest time resolution, i.e. $\Delta t = 100$, we find that the L2 norm decreases of the 2 orders of magnitude as the solver's

accuracy increases by 1 order. This also means that fixing a tolerance on the errors, the higher is the solver's accuracy the larger is the time resolution available. As an example, assuming that admissible errors are of $O(10^{-2})$ with the 4-steps solver we can use $\Delta t = 625$ performing $N_s = t_final/625$ numerical integration steps, whereas using a 3-steps solvers we must adopt $\Delta t = 100$ performing $6.25 \times N_s$ numerical integration steps instead of N_s. Considering that the computational costs is only slightly affected by the number of previous time steps considered[8], the accuracy order has strong impact on the overall numerical efficiency: to improve the numerical efficiency reducing the computational costs, the usage of high order Adams-Bashforth solvers with larger time steps should be preferred instead of low order solvers with smaller time steps.

Table 3: Oscillation test: errors analysis of explicit Adams-Bashforth solvers

(a) 1 step

Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.840E+10	0.706E+10	/	/
2500.0	0.503E+06	0.570E+06	14.03	13.60
1250.0	0.289E+04	0.272E+04	7.45	7.71
625.0	0.239E+03	0.232E+03	3.59	3.55
320.0	0.737E+02	0.722E+02	1.76	1.74
100.0	0.250E+02	0.247E+02	0.93	0.92

(b) 2 steps

Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.596E+03	0.583E+03	/	/
2500.0	0.221E+02	0.218E+02	4.75	4.74
1250.0	0.764E+01	0.769E+01	1.53	1.50
625.0	0.265E+01	0.268E+01	1.53	1.52
320.0	0.968E+00	0.981E+00	1.51	1.50
100.0	0.169E+00	0.171E+00	1.50	1.50

(c) 3 steps

Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.857E+01	0.854E+01	/	/
2500.0	0.391E+01	0.386E+01	1.13	1.14
1250.0	0.825E+00	0.814E+00	2.24	2.25
625.0	0.150E+00	0.148E+00	2.46	2.46
320.0	0.282E-01	0.278E-01	2.49	2.49
100.0	0.154E-02	0.152E-02	2.50	2.50

(d) 4 steps

Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.128E+07	0.143E+07	/	/
2500.0	0.106E+01	0.107E+01	20.21	20.34
1250.0	0.967E-01	0.981E-01	3.45	3.45
625.0	0.859E-02	0.871E-02	3.49	3.49
320.0	0.827E-03	0.838E-03	3.50	3.50
100.0	0.141E-04	0.143E-04	3.50	3.50

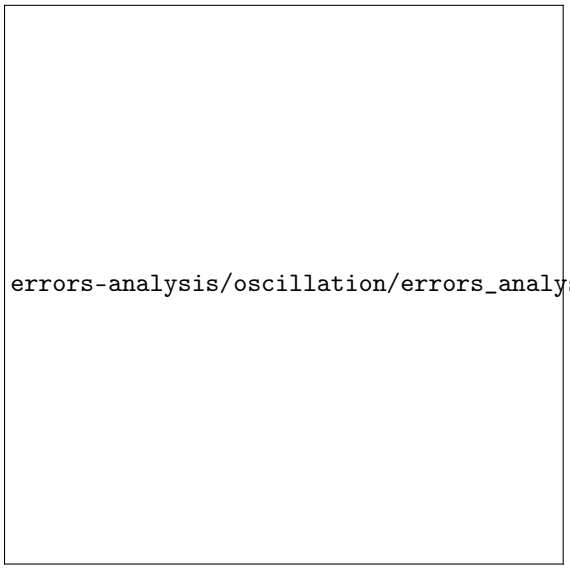
Figure 1 shows, for each solver exercised, the $X(t)$ and $Y(t)$ solution for $t \in [0, 10^6]$: the plots into the figure report a global overview of the solution for all the instants considered (left subplots) and a detailed zoom over the last instants of the integration (right subplots) for evaluating the numerical errors accumulation. For the sake of clarity, the strongly unstable solutions are omitted into the subplots concerning the final integration instants, namely the solutions for large Δt. Figure 1 emphasizes the instability generation for some pairs steps number/Δt. The 2 and 4 steps solutions are instable for $\Delta t = 5000 \rightarrow f * \Delta t = 0.5$. On the contrary, the 3 steps solution is stable, but the amplitude is dumped and the solution vanishes as the integration proceeds. The 2 and 4 steps solutions show a phase error that decreases as the time resolution increases, whereas 3 steps solution has null phase error.

4.1.4. Adams-Bashforth-Moulton

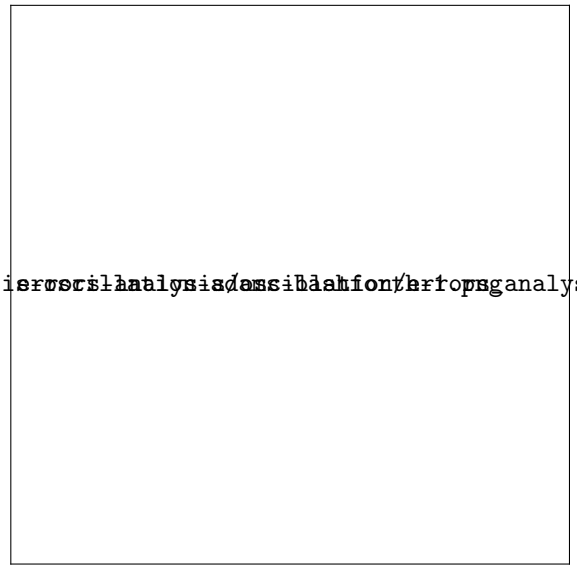
Table 4 summarizes the Adams-Bashforth-Moulton error analysis. The same considerations done for the Adams-Bashforth solutions can repeated for the Adams-Bashforth-Moulton ones, thus they are omitted for the sake of conciseness. An interesting result concerns the observed errors: the $O(10^{-2})$ error is now obtained with $\Delta t = 1250$ for the 4-steps solver, thus it is 2 times faster than the corresponding Adams-Bashforth 4-step solver. Considering that the computational costs of a single Adams-Bashforth-Moulton step is only slightly greater than the corresponding Adamas-Bashforth step, the efficiency increasing is not negligible.

Figure 2 shows similar plots of figure 1 above discussed. Differently from the Adams-Bashforth class, the amplitude damping feature is now possessed by the 2-steps solver, see plot 2B, while all solutions show phase errors that decrease as the time resolution increases.

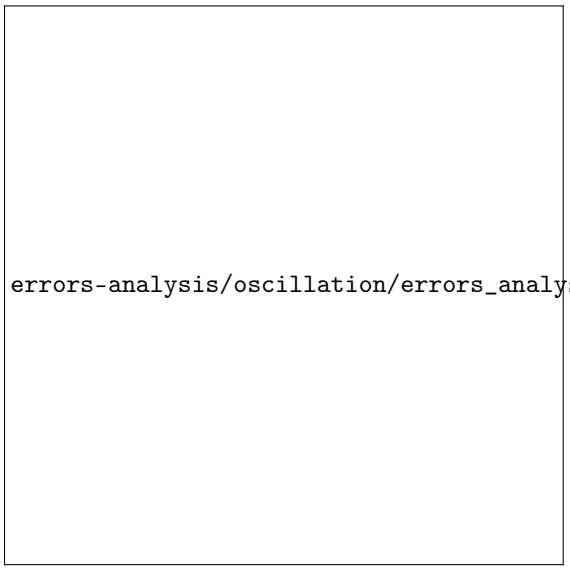
[8]Recalling equation 3 one can observe that there is only one new evaluation of the residuals function R independently of the previous time steps considered. Thus, the computational costs is affected only by the increasing number of residuals summations, the costs of which are typically negligible with respect the cost of R evaluation.



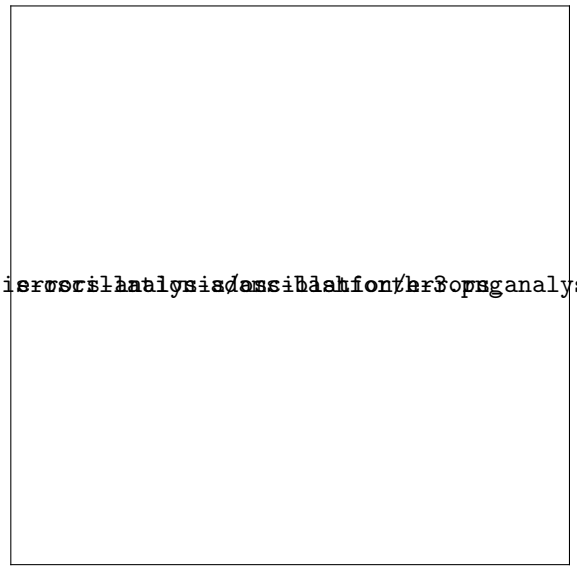
(A) 1 step



(B) 2 steps



(C) 3 steps



(D) 4 steps

Figure 1: Oscillation equations solutions computed by means of Adams-Bashforth solvers

Table 4: Oscillation test: errors analysis of predictor-corrector Adams-Bashforth-Moulton solvers

(a) 1 step

Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.241E+20	0.266E+20	/	/
2500.0	0.664E+11	0.716E+11	28.44	28.47
1250.0	0.952E+06	0.100E+07	16.09	16.12
625.0	0.413E+04	0.407E+04	7.85	7.95
320.0	0.387E+03	0.383E+03	3.54	3.53
100.0	0.145E+03	0.145E+03	0.84	0.83

(b) 2 steps

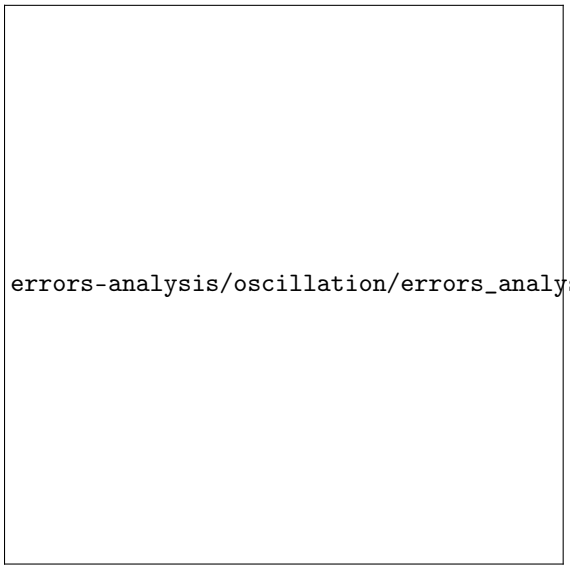
Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.704E+01	0.701E+01	/	/
2500.0	0.392E+01	0.395E+01	0.84	0.83
1250.0	0.148E+01	0.150E+01	1.40	1.39
625.0	0.526E+00	0.534E+00	1.49	1.49
320.0	0.193E+00	0.196E+00	1.50	1.50
100.0	0.338E-01	0.342E-01	1.50	1.50

(c) 3 steps

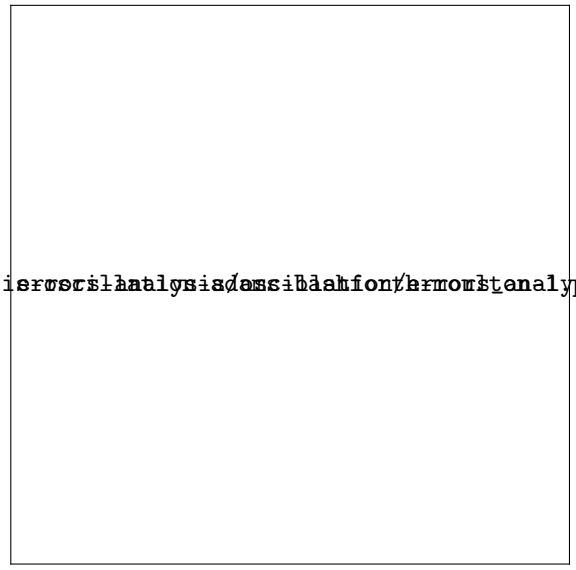
Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.457E+01	0.464E+01	/	/
2500.0	0.656E+00	0.654E+00	2.80	2.83
1250.0	0.100E+00	0.987E-01	2.71	2.73
625.0	0.169E-01	0.167E-01	2.56	2.56
320.0	0.314E-02	0.310E-02	2.52	2.51
100.0	0.171E-03	0.169E-03	2.50	2.50

(d) 4 steps

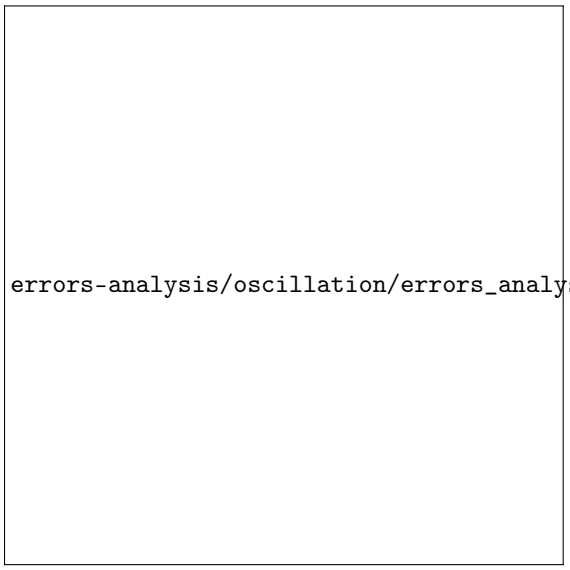
Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.229E+01	0.225E+01	/	/
2500.0	0.119E+00	0.118E+00	4.26	4.25
1250.0	0.825E-02	0.833E-02	3.85	3.83
625.0	0.671E-03	0.681E-03	3.62	3.61
320.0	0.631E-04	0.640E-04	3.53	3.53
100.0	0.107E-05	0.108E-05	3.51	3.51



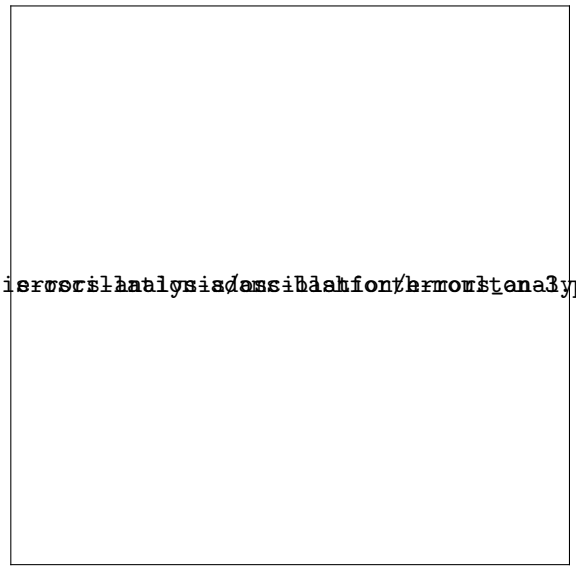
(A) 1 step



(B) 2 steps



(C) 3 steps



(D) 4 steps

Figure 2: Oscillation equations solutions computed by means of Adams-Bashforth-Moulton solvers

4.1.5. Adams-Moulton

Table 5 summarizes the Adams-Moulton error analysis. The implicit Adams-Moulton solvers behave much like the Adams-Bashforth-Moulton ones: they have similar errors and observed orders for the same formal order considered. However, the implicit Adams-Moulton class uses one less step with respect the corresponding Adams-Bashforth-Moulton class: this could lead to the promise of higher computational efficiency. Notwithstanding, for solving the implicit non-linearity embedded into the Adams-Moulton solvers an iterative algorithm must be employed: for the results presented, a 5 iterations of *fixed point algorithm* have been computed. This strongly reduces the eventual gain of computational efficiency with respect the Adams-Bashforth-Moulton class.

Table 5: Oscillation test: errors analysis of explicit Adams-Moulton solvers; the implicit non-linearity is solved by 5 iterations of *fixed point algorithm*

(a) 1 step

Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.840E+10	0.706E+10	/	/
2500.0	0.503E+06	0.570E+06	14.03	13.60
1250.0	0.289E+04	0.272E+04	7.45	7.71
625.0	0.239E+03	0.232E+03	3.59	3.55
320.0	0.737E+02	0.722E+02	1.76	1.74
100.0	0.250E+02	0.247E+02	0.93	0.92

(b) 2 steps

Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.108E+02	0.109E+02	/	/
2500.0	0.412E+01	0.419E+01	1.39	1.38
1250.0	0.148E+01	0.150E+01	1.48	1.48
625.0	0.527E+00	0.533E+00	1.49	1.49
320.0	0.193E+00	0.196E+00	1.50	1.50
100.0	0.338E-01	0.342E-01	1.50	1.50

(c) 3 steps

Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.390E+01	0.384E+01	/	/
2500.0	0.551E+00	0.544E+00	2.82	2.82
1250.0	0.947E-01	0.934E-01	2.54	2.54
625.0	0.167E-01	0.165E-01	2.50	2.50
320.0	0.313E-02	0.309E-02	2.50	2.50
100.0	0.171E-03	0.169E-03	2.50	2.50

(d) 4 steps

Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.983E+00	0.999E+00	/	/
2500.0	0.832E-01	0.845E-01	3.56	3.56
1250.0	0.736E-02	0.746E-02	3.50	3.50
625.0	0.652E-03	0.660E-03	3.50	3.50
320.0	0.626E-04	0.635E-04	3.50	3.50
100.0	0.107E-05	0.108E-05	3.50	3.50

Figure 3 shows similar plots of figure 2 above discussed: there are not relevant differences between the 2 classes of solvers.

4.1.6. Leapfrog

The Leapfrog solutions are in agreement with the expected results: both unfiltered and RAW-filtered solutions show an observed order of accuracy that tends to the formal 2^{nd} order. The two solutions are almost the same.

Table 6: Oscillation test: errors analysis of explicit Leapfrog solvers

(a) Unfiltered

Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.156E+02	0.156E+02	/	/
2500.0	0.849E+01	0.846E+01	0.87	0.88
1250.0	0.300E+01	0.303E+01	1.50	1.48
625.0	0.106E+01	0.107E+01	1.51	1.50
320.0	0.387E+00	0.392E+00	1.50	1.50
100.0	0.676E-01	0.685E-01	1.50	1.50

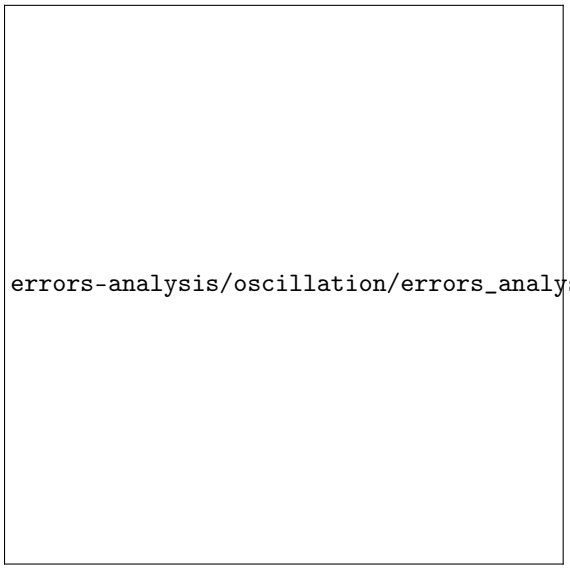
(b) RAW-filtered

Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.156E+02	0.156E+02	/	/
2500.0	0.855E+01	0.852E+01	0.86	0.87
1250.0	0.303E+01	0.305E+01	1.50	1.48
625.0	0.107E+01	0.108E+01	1.51	1.50
320.0	0.390E+00	0.395E+00	1.50	1.50
100.0	0.685E-01	0.692E-01	1.50	1.50

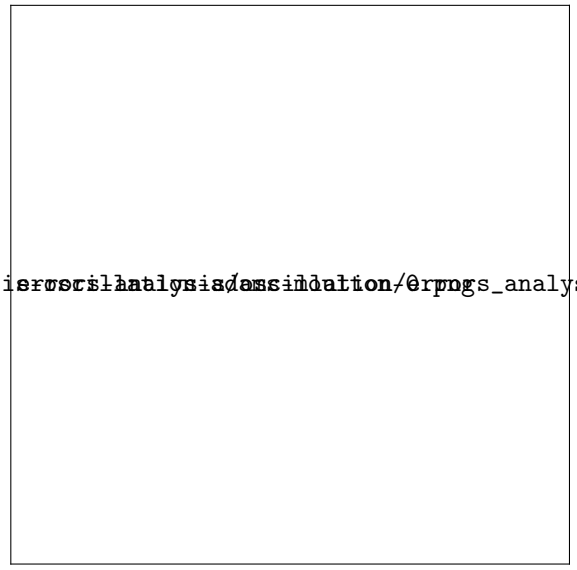
4.1.7. Low Storage Runge-Kutta

4.1.8. TVD/SSP Runge-Kutta

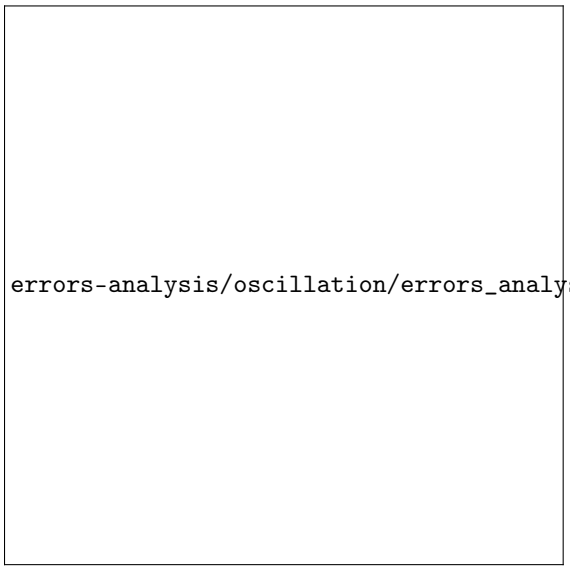
4.1.9. Embedded Runge-Kutta



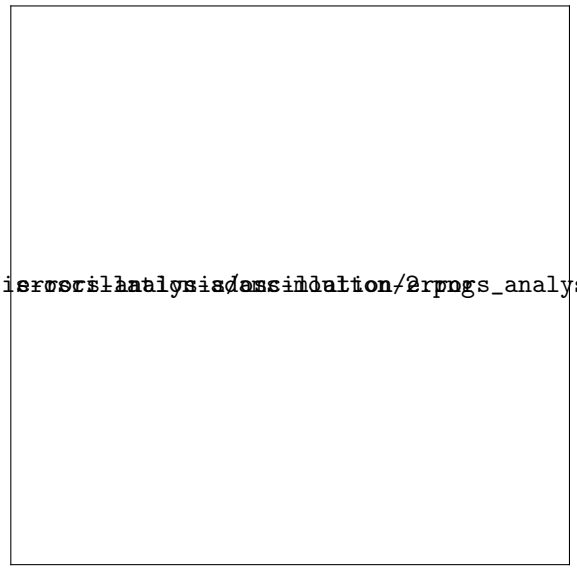
(A) 0 step



(B) 1 step



(C) 2 steps



(D) 3 steps

Figure 3: Oscillation equations solutions computed by means of Adams-Moulton solvers

(A) Unfiltered (B) RAW-filtered

Figure 4: Oscillation equations solutions computed by means of Leapfrog solvers

Table 7: Oscillation test: errors analysis of explicit Low Storage Runge-Kutta solvers

(a) 1 stage

Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.840E+10	0.706E+10	/	/
2500.0	0.503E+06	0.570E+06	14.03	13.60
1250.0	0.289E+04	0.272E+04	7.45	7.71
625.0	0.239E+03	0.232E+03	3.59	3.55
320.0	0.737E+02	0.722E+02	1.76	1.74
100.0	0.250E+02	0.247E+02	0.93	0.92

(b) 5 stages

Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.120E+00	0.122E+00	/	/
2500.0	0.106E-01	0.107E-01	3.51	3.51
1250.0	0.935E-03	0.947E-03	3.50	3.50
625.0	0.826E-04	0.836E-04	3.50	3.50
320.0	0.793E-05	0.803E-05	3.50	3.50
100.0	0.135E-06	0.137E-06	3.50	3.50

(c) 6 stages

Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.979E-01	0.994E-01	/	/
2500.0	0.876E-02	0.888E-02	3.48	3.48
1250.0	0.776E-03	0.786E-03	3.50	3.50
625.0	0.686E-04	0.695E-04	3.50	3.50
320.0	0.659E-05	0.667E-05	3.50	3.50
100.0	0.112E-06	0.114E-06	3.50	3.50

(d) 7 stages

Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.238E-01	0.240E-01	/	/
2500.0	0.203E-02	0.205E-02	3.55	3.55
1250.0	0.177E-03	0.180E-03	3.51	3.51
625.0	0.156E-04	0.158E-04	3.50	3.50
320.0	0.150E-05	0.152E-05	3.50	3.50
100.0	0.269E-07	0.273E-07	3.46	3.46

(e) 12 stages

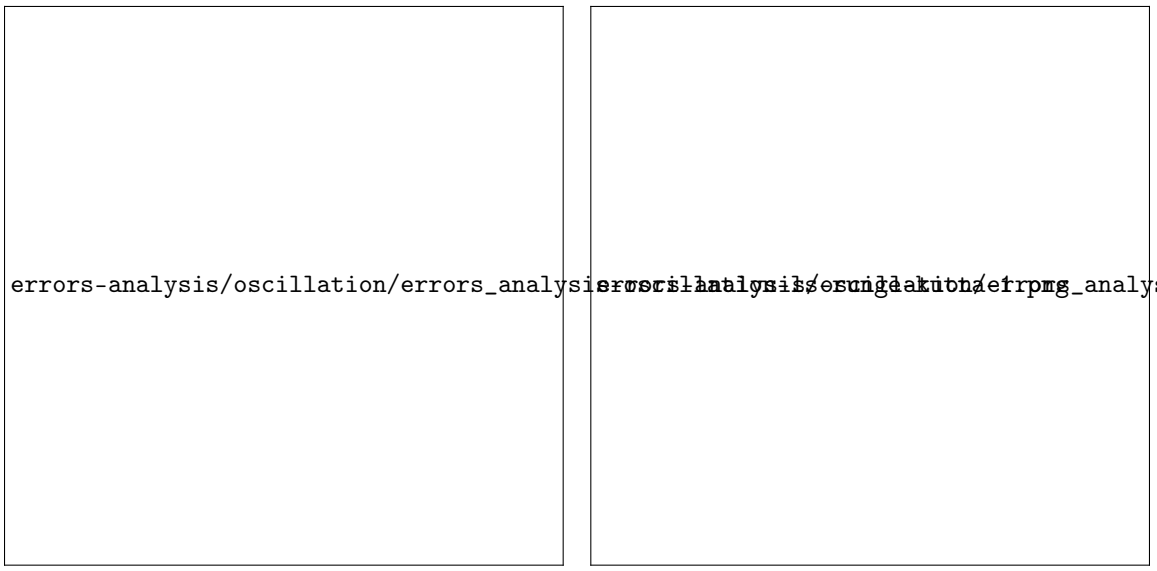
Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.195E-01	0.198E-01	/	/
2500.0	0.175E-02	0.177E-02	3.48	3.48
1250.0	0.155E-03	0.157E-03	3.50	3.50
625.0	0.137E-04	0.139E-04	3.50	3.50
320.0	0.132E-05	0.133E-05	3.50	3.50
100.0	0.225E-07	0.228E-07	3.50	3.50

(f) 13 stages

Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.795E-02	0.805E-02	/	/
2500.0	0.703E-03	0.712E-03	3.50	3.50
1250.0	0.621E-04	0.629E-04	3.50	3.50
625.0	0.549E-05	0.556E-05	3.50	3.50
320.0	0.527E-06	0.534E-06	3.50	3.50
100.0	0.899E-08	0.911E-08	3.50	3.50

(g) 14 stages

Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.849E-02	0.860E-02	/	/
2500.0	0.750E-03	0.759E-03	3.50	3.50
1250.0	0.662E-04	0.671E-04	3.50	3.50
625.0	0.585E-05	0.593E-05	3.50	3.50
320.0	0.562E-06	0.569E-06	3.50	3.50
100.0	0.959E-08	0.972E-08	3.50	3.50



(A) 1 stage (B) 5 stages

Figure 5: Oscillation equations solutions computed by means of low storage Runge-Kutta solvers

Table 8: Oscillation test: errors analysis of explicit TVD/SSP Runge-Kutta

(a) 1 stage

Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.840E+10	0.706E+10	/	/
2500.0	0.503E+06	0.570E+06	14.03	13.60
1250.0	0.289E+04	0.272E+04	7.45	7.71
625.0	0.239E+03	0.232E+03	3.59	3.55
320.0	0.737E+02	0.722E+02	1.76	1.74
100.0	0.250E+02	0.247E+02	0.93	0.92

(b) 2 stages

Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.316E+02	0.319E+02	/	/
2500.0	0.892E+01	0.894E+01	1.83	1.84
1250.0	0.301E+01	0.305E+01	1.57	1.55
625.0	0.106E+01	0.107E+01	1.51	1.51
320.0	0.387E+00	0.392E+00	1.50	1.50
100.0	0.676E-01	0.685E-01	1.50	1.50

(c) 3 stages

Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.255E+01	0.252E+01	/	/
2500.0	0.523E+00	0.516E+00	2.28	2.29
1250.0	0.944E-01	0.931E-01	2.47	2.47
625.0	0.167E-01	0.165E-01	2.50	2.50
320.0	0.314E-02	0.310E-02	2.50	2.50
100.0	0.171E-03	0.169E-03	2.50	2.50

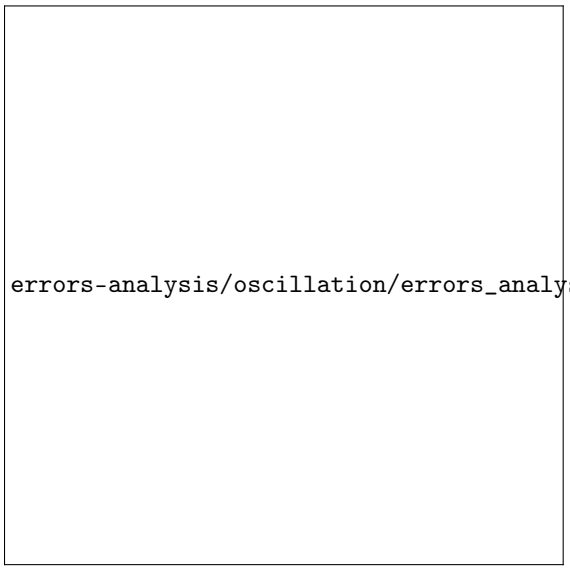
(d) 5 stages

Time Step	Error X	Error Y	Order X	Order Y
5000.0	0.139E+00	0.141E+00	/	/
2500.0	0.122E-01	0.124E-01	3.50	3.50
1250.0	0.108E-02	0.110E-02	3.50	3.50
625.0	0.956E-04	0.969E-04	3.50	3.50
320.0	0.937E-05	0.949E-05	3.47	3.47
100.0	0.512E-06	0.519E-06	2.50	2.50

Table 9: Oscillation test: errors analysis of explicit embedded Runge-Kutta solvers

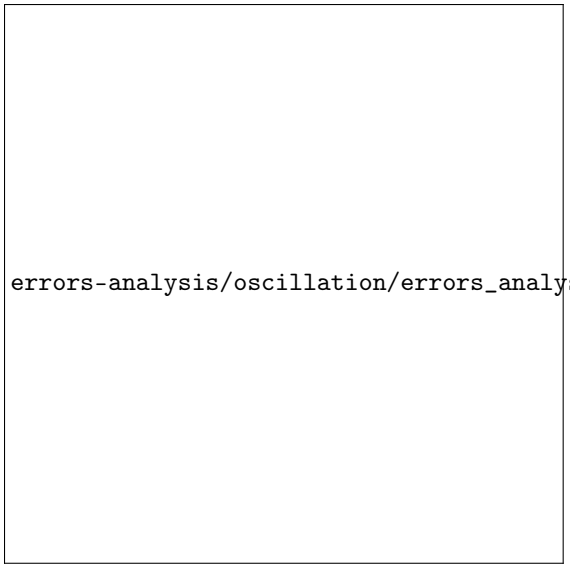
(a) 7 stages

Mean Time Step	Error X	Error Y	Order X	Order Y
7352.9	0.626E-01	0.622E-01	/	/
3759.4	0.482E-02	0.480E-02	3.82	3.82
2272.7	0.793E-03	0.786E-03	3.59	3.59
1379.3	0.393E-04	0.390E-04	6.02	6.02
618.8	0.216E-05	0.213E-05	3.62	3.62
356.0	0.252E-06	0.249E-06	3.89	3.88

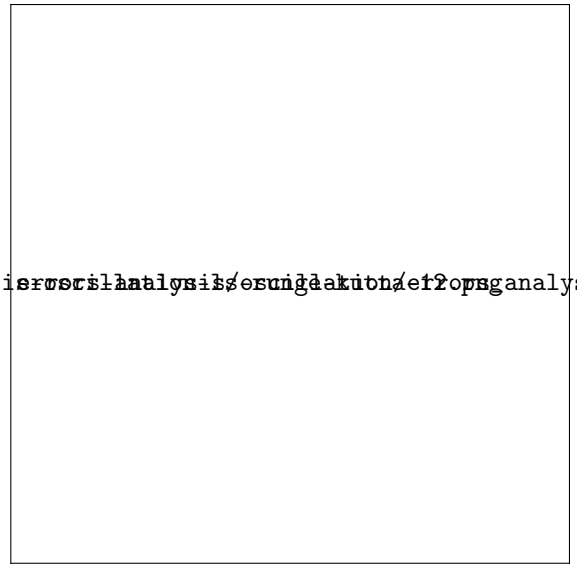


(A) 6 stages

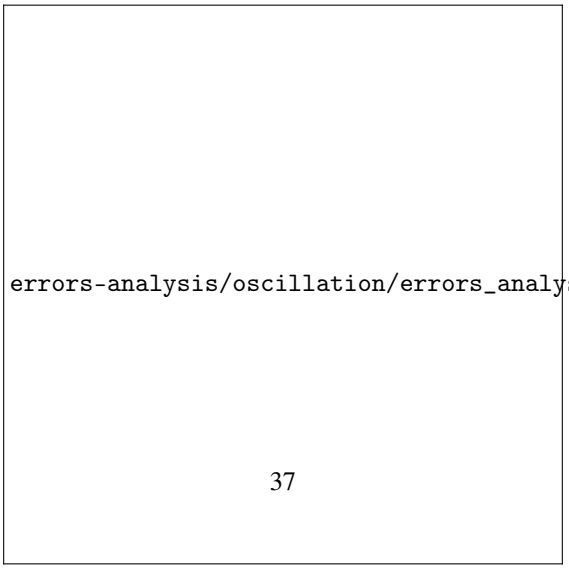
(B) 7 stages



(C) 12 stages

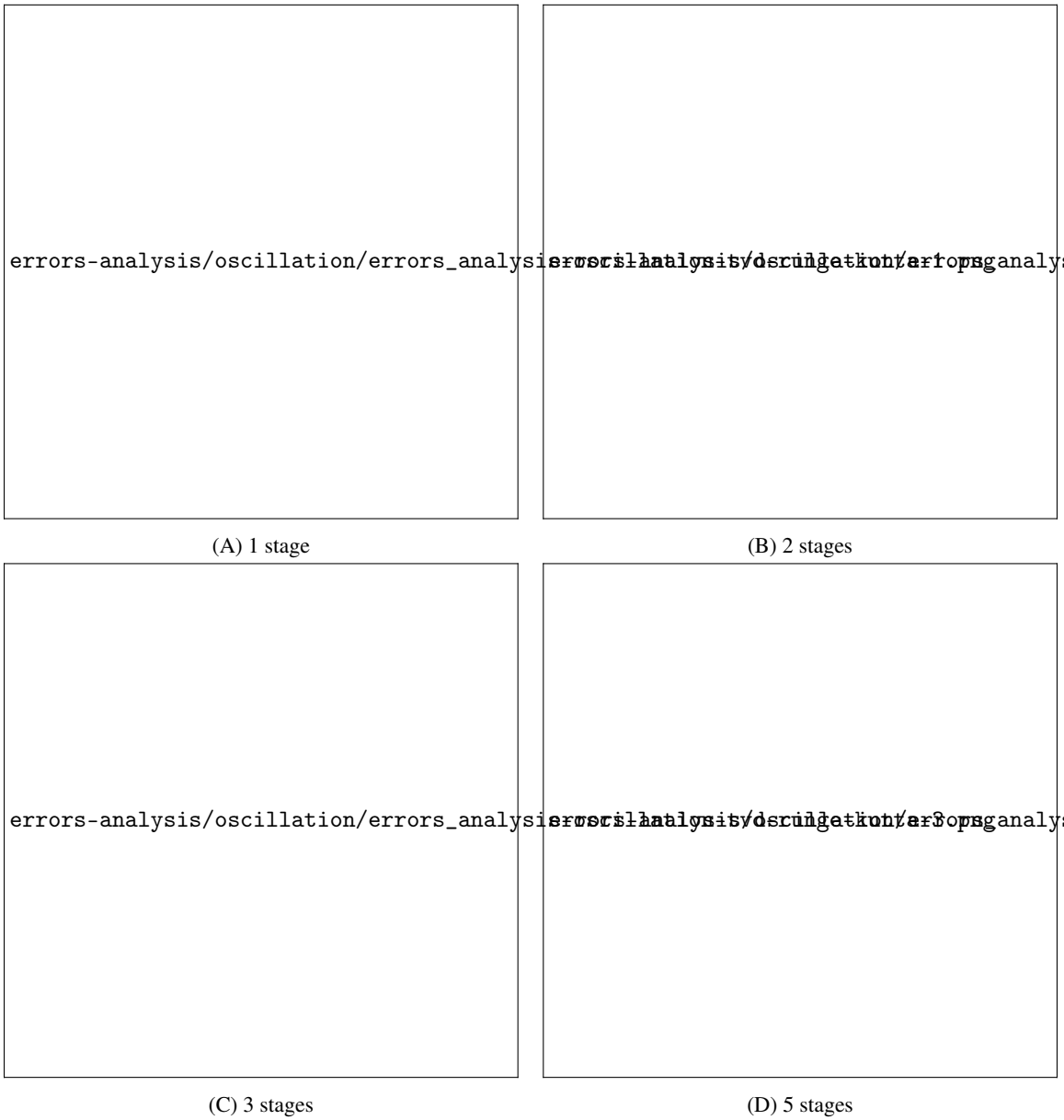


(D) 13 stages



(E) 14 stages

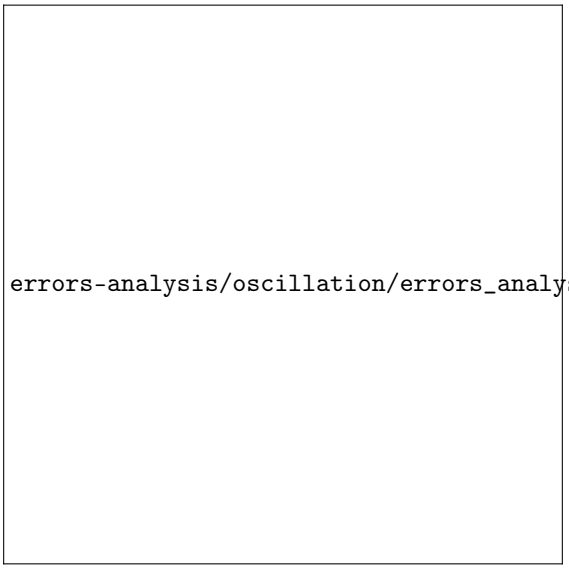
Figure 6: Oscillation equations solutions computed by means of low storage Runge-Kutta solvers



(A) 1 stage (B) 2 stages

(C) 3 stages (D) 5 stages

Figure 7: Oscillation equations solutions computed by means of TVD/SSP Runge-Kutta solvers



(A) 7 stages

Figure 8: Oscillation equations solutions computed by means of embedded Runge-Kutta solvers

5. Benchmarks on parallel frameworks

As aforementioned, FOODIE is unaware of any parallel paradigms or programming models the client codes adopt. As a consequence, the parallel performances measurements presented into this section are aimed only to prove that FOODIE environment does not destroy the parallel scaling of the baseline code implemented without FOODIE.

To obtain such a prove, the 1D Euler PDE system described previously is numerically solved with FOODIE-aware test codes that in turn exploit parallel resources by means:

- CoArray Fortran (CAF) model, for shared and distributed memory architectures;
- OpenMP directive-based model, for only shared memory architectures;

In order to measure the performances of the parallel-enabled FOODIE tests, the *strong* and *weak* scaling have been considered. For the strong scaling the *speedup* has been computed:

$$speedup(N,k) = \frac{T_{serial}(N)}{T_{parallel}(N,k)} \tag{20}$$

where N is the problem size, K the number of parallel resources used (namely the physical cores), T_{serial} is the CPU time of the serial code and $T_{parallel}$ the one of the parallel code. The ideal speedup is linear with slop equals to 1. The efficiency correlated to the strong scaling measurement is defined as:

$$efficiency(N,k) = \frac{speedup(N,k)}{k} \tag{21}$$

The maximum ideal efficiency is obviously the unity.

For the of weak scaling measurement the *sizeup* has been computed:

$$sizeup(N,k) = \frac{N_k}{N_1} \cdot \frac{T_{serial}(N_1)}{T_{parallel}(N_k,k)} \tag{22}$$

where N_1 is the minimum size considered and N_K is the size used for the test computed with k parallel resources. If N_K is scaled proportional to N_1, the ideal sizeup is again linear and if $N_k = k \cdot N_1$ the slope is again linear. The efficiency correlated to the weak scaling is defined as:

$$efficiency(N,k) = \frac{sizeup(N,k)}{k} \tag{23}$$

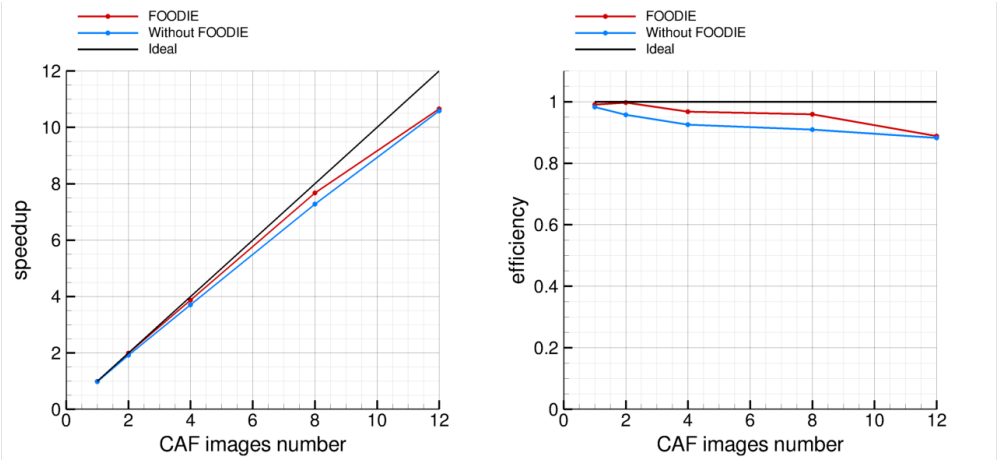
The maximum ideal efficiency is obviously the unity.

The same 1D Euler PDEs problem is also solved by parallel-enabled codes that are not based on FOODIE: their solutions provide a reference for measuring the effect of FOODIE abstraction on the parallel scaling.

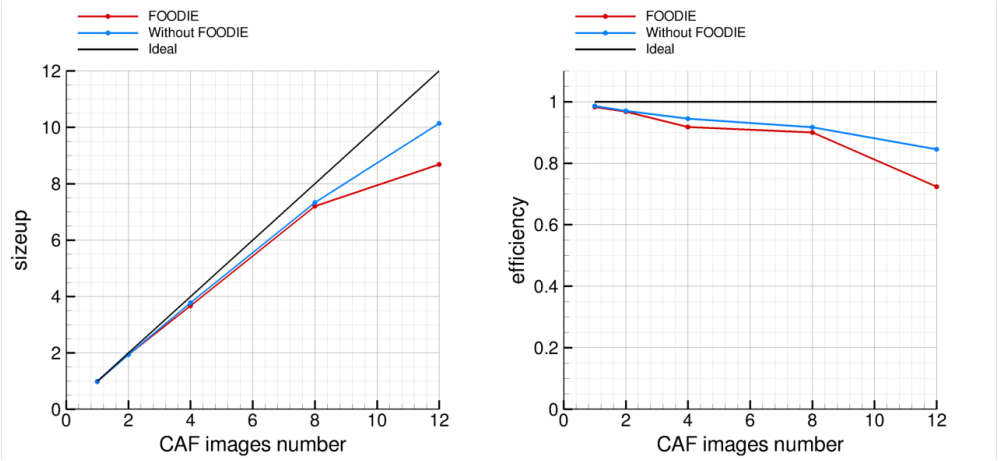
5.1. CAF benchmark

This subsection reports the parallel scaling analysis of Euler 1D test programs (with and without FOODIE) being parallelized by means of CoArrays Fortran (CAF) model. This parallel model is based on the concept of *coarray* introduced into the Fortran 2008 standard: the array syntax is extended introducing the so called *codimension* that is a new arrays indexing. Essentially, a CAF enabled code is designed to be replicated a certain number of times and all copies, conventionally named *images*, are executed asynchronously. Each image has its own set of data (memory) and the codimension indexes are used to access to the (remote) memory of the other images. The CAF approach allows the implementation of Partitioned Global Address Space (PGAS) model following the SPMD (single program, multiple data) parallelization paradigm. The programmer must take care of defining the coarray variables and of synchronizing the images when necessary. This approach requires the refactoring of legacy serial codes, but it allows the exploitation of both shared and distributed memory architectures. Moreover, it is a standard feature of Fortran (2008), thus it is not chained to any particular compiler vendors extension.

The benchmarks shows in this section have been done on a *dual Intel(R) Xeon(R) CPU X5650* exacores workstation for a total of 12 physical cores, coupled with 24GB of RAM. In order to perform an accurate analysis 4 different codes have considered:



(A) Strong scaling, number of cells 240000



(B) Weak scaling, minimum number of cells 24000

Figure 9: Scaling efficiency with CAF programming model

- FOODIE-aware codes:
  - serial code;
  - CAF-enabled code;
- procedural codes without using FOODIE library:
  - serial code;
  - CAF-enabled code;

These codes (see Appendix B.2 for the implementation details) have been compiled by means of the GNU gfortran compiler v5.2.0 coupled with OpenCoarrays v1.1.0[9].

The Euler conservation laws are integrated for 30 time steps by means of the TVD RK(5,4) solver: the measured CPU time used for computing the scaling efficiencies is the average of the 30 integrations, thus representing the mean CPU time for computing one time step integration.

[9]OpenCoarrays is an open-source software project for developing, porting and tuning transport layers that support coarray Fortran (CAF) compilers, see [**?**].

For the strong scaling, the benchmark has been conducted with 240000 finite volumes. Figure 9A summarizes the strong scaling analysis: it shows that FOODIE-based code scales similarly to the baseline code without FOODIE.

For the weak scaling the minimum size is 24000 finite volumes and the size is scaled linearly with the CAF images, thus $N_{12} = 288000$ cells. Figure 9B summarizes the weak scaling analysis and it essentially confirms that FOODIE-based code scales similarly to the baseline code without FOODIE.

Both strong and weak scaling analysis point out that for the computing architecture considered the parallel scaling is reasonable up to 12 cores, the efficiency being always satisfactory.

To complete the comparison, the absolute CPU-time consumed by the two families of codes (with and without FOODIE) must be considered. Table 10 summarizes the benchmarks results. As shown, procedural and FOODIE-aware codes consume a very similar CPU-time for both the strong and the weak benchmarks. The same results are shown in figure 10. These results prove that the abstraction of FOODIE environment does not degrade the computational efficiency.

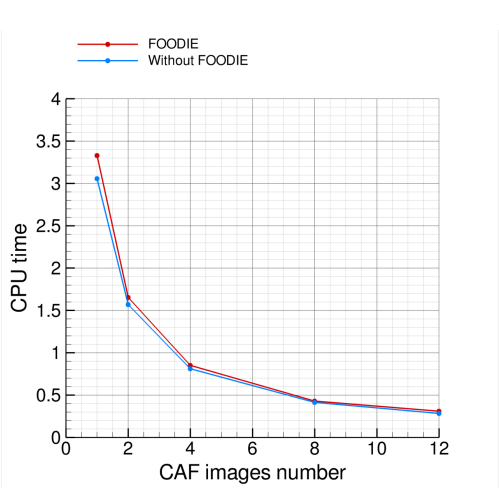
Table 10: caf benchmarks results

(a) Strong benchmarks, number of cells 240000

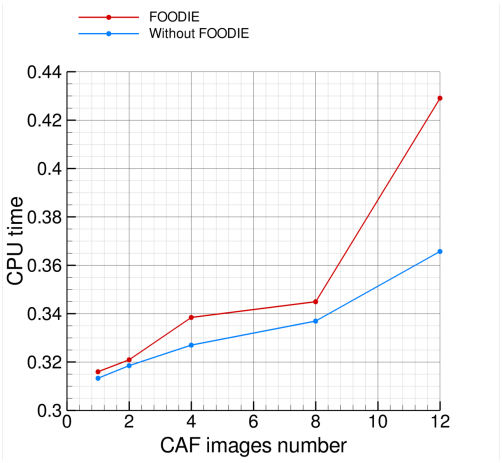
Number of caf threads	CPU time for 1 time step integration			
	FOODIE serial	FOODIE parallel	procedural serial	procedural parallel
1	3.2970	3.3297	3.0049	3.0563
2	/	1.6536	/	1.5686
4	/	0.8515	/	1.8116
8	/	0.4296	/	0.4130
12	/	0.3094	/	0.2839

(b) Weak benchmarks, minimum number of cells 24000

Number of caf threads	Number of Cells	CPU time for 1 time step integration			
		FOODIE serial	FOODIE parallel	procedural serial	procedural parallel
1	24000	0.3105	0.3159	0.3089	0.3133
2	48000	/	0.3209	/	0.3185
4	96000	/	0.3384	/	0.3269
8	192000	/	0.3449	/	0.3369
12	288000	/	0.4291	/	0.3657



(A) Strong benchmark, number of cells 240000

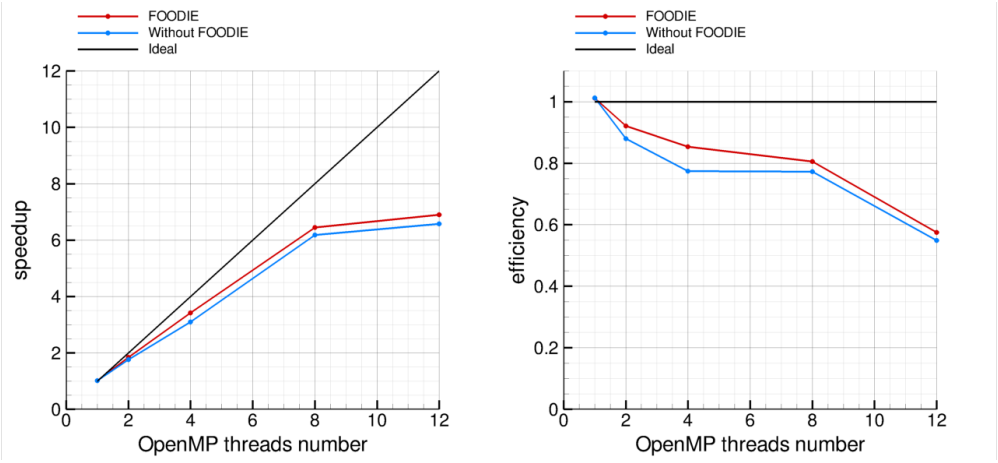


(B) Weak benchmark, minimum number of cells 24000

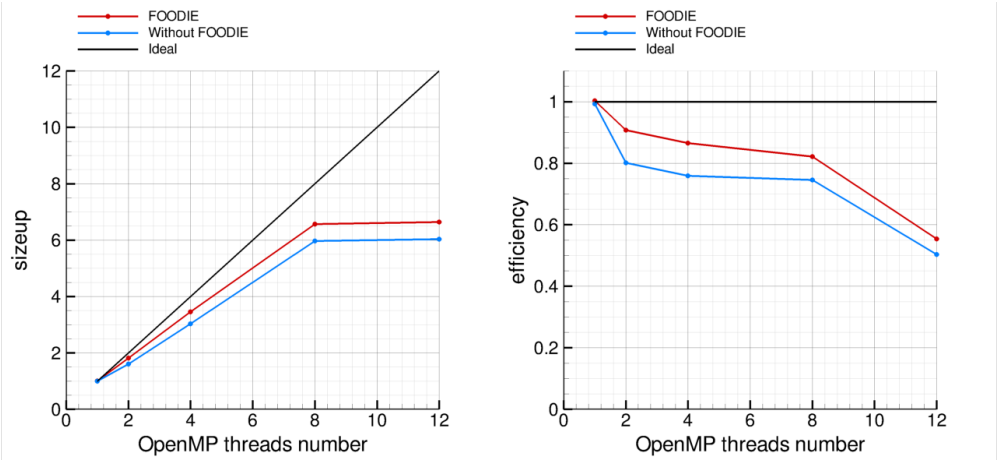
Figure 10: CPU time consumed with caf programming model

5.2. *OpenMP benchmark*

This subsection reports the parallel scaling analysis of Euler 1D test programs (with and without FOODIE) being parallelized by means of OpenMP directives-based paradigm. This parallel model is based on the concept of *threads*: an OpenMP enabled code start a single (master) threaded program and, at run-time, it is able to generate a team of (many) threads that work concurrently on the parallelized parts of the code, thus reducing the CPU time necessary for completing such parts. The parallelization is made by means of *directives* explicitly inserted by the programmer: the communications between threads are automatically handled by the compiler (through the provided OpenMP library used as back-end). OpenMP parallel paradigm is not a standard feature of Fortran, rather it is an extension provided by the compiler vendors. This parallel paradigm constitutes an effective and easy approach for parallelizing legacy serial codes, however its usage is limited to shared memory architectures because all threads must have access to the same memory.



(A) Strong scaling, number of cells 240000



(B) Weak scaling, minimum number of cells 24000

Figure 11: Scaling efficiency with OpenMP programming model

The benchmarks shown in this section have been done on a *dual Intel(R) Xeon(R) CPU X5650* exacores workstation for a total of 12 physical cores, coupled with 24GB of RAM. In order to perform an accurate analysis 4 different codes have considered:

- FOODIE-aware codes:

  - serial code;
  - OpenMP-enabled code;

- procedural codes without using FOODIE library:
  - serial code;
  - OpenMP-enabled code;

These codes (see Appendix B.3 for the implementation details) have been compiled by means of the GNU gfortran compiler v5.2.0 with *-O2 -fopenmp* compilation flags.

The Euler conservation laws are integrated for 30 time steps by means of the TVD RK(5,4) solver: the measured CPU time used for computing the scaling efficiencies is the average of the 30 integrations, thus representing the mean CPU time for computing one time step integration.

For the strong scaling, the benchmark has been conducted with 240000 finite volumes. Figure 11A summarizes the strong scaling analysis: it shows that FOODIE-based code scales similarly to the baseline code without FOODIE.

For the weak scaling the minimum size is 24000 finite volumes and the size is scaled linearly with the OpenMP threads, thus $N_{12} = 288000$ cells. Figure 11B summarizes the weak scaling analysis and it essentially confirms that FOODIE-based code scales similarly to the baseline code without FOODIE.

Both strong and weak scaling analysis point out that for the computing architecture considered the parallel scaling is reasonable up to 8 cores: using 12 cores the measured efficiencies become unsatisfactory, reducing below the 60%.

To complete the comparison, the absolute CPU-time consumed by the two families of codes (with and without FOODIE) must be considered. Table 11 summarizes the benchmarks results. As shown, procedural and FOODIE-aware codes consume a very similar CPU-time for both the strong and the weak benchmarks. The same results are shown in figure 12. These results prove that the abstraction of FOODIE environment does not degrade the computational efficiency.

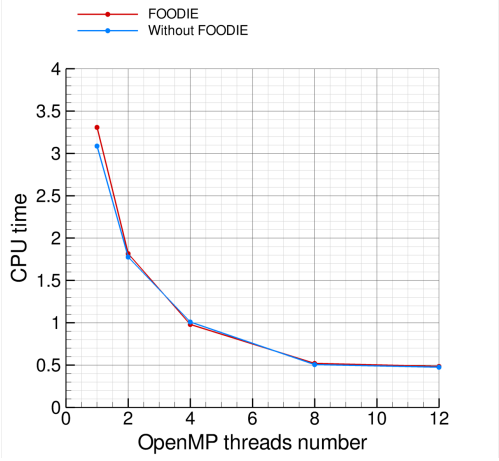
Table 11: OpenMP benchmarks results

(a) Strong benchmarks, number of cells 240000

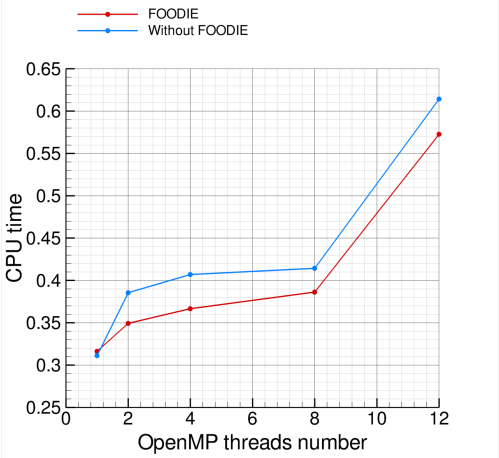
Number of OpenMP threads	CPU time for 1 time step integration			
	FOODIE serial	FOODIE parallel	procedural serial	procedural parallel
1	3.3466	3.3076	3.1252	3.0873
2	/	1.8166	/	1.7765
4	/	0.9798	/	1.0085
8	/	0.5192	/	0.5055
12	/	0.4847	/	0.4748

(b) Weak benchmarks, minimum number of cells 24000

Number of OpenMP threads	Number of Cells	CPU time for 1 time step integration			
		FOODIE serial	FOODIE parallel	procedural serial	procedural parallel
1	24000	0.3171	0.3162	0.3089	0.3111
2	48000	/	0.3492	/	0.3854
4	96000	/	0.3666	/	0.4069
8	192000	/	0.3862	/	0.4142
12	288000	/	0.5727	/	0.6142



(A) Strong benchmark, number of cells 240000



(B) Weak benchmark, minimum number of cells 24000

Figure 12: CPU time consumed with OpenMP programming model

6. Concluding Remarks and Perspectives

To be written.

Appendix A. Solvers details

Table A.12: Explicit Adams-Bashforth coefficients

N_s	b_1	b_2	b_3	b_4
2	$-\frac{1}{2}$	$\frac{3}{2}$	/	/
3	$\frac{5}{12}$	$-\frac{16}{12}$	$\frac{23}{12}$	/
4	$-\frac{9}{24}$	$\frac{37}{24}$	$-\frac{59}{24}$	$\frac{55}{24}$

Table A.13: Butcher's table of 2 stages, 2^{nd} order, Runge-Kutta TVD scheme

1	1	0
	1/2	1/2

Table A.14: Butcher's table of 3 stages, 3^{rd} order, Runge-Kutta SSP scheme

1	1		
1/2	1/4	1/4	
	1/6	1/6	2/3

Table A.15: Butcher's table of 5 stages, 4^{th} order accurate, Runge-Kutta SSP scheme

0.39175222700392	0.39175222700392				
0.58607968896779	0.21766909633821	0.36841059262959			
0.47454236302687	0.08269208670950	0.13995850206999	0.25189177424738		
0.93501063100924	0.06796628370320	0.11503469844438	0.20703489864929	0.54497475021237	
	0.14681187618661	0.24848290924556	0.10425883036650	0.27443890091960	0.22600748319395

Appendix B. Euler 1D Parallel Tests API

In subsections 5.1 and 5.2 it has been proved that FOODIE usage does not penalize the parallel scaling of an equivalent procedural code implemented without FOODIE. To this aim, we have solved the Euler's conservation laws (in one dimension) by means of FOODIE: as a matter of fact, Euler 1D PDEs constitutes a complex test retaining many difficulties of real applications, but it is still simple enough to serve as benchmark test. In this section we report the implementation details of the codes developed to solve (with serial and parallel models) the Euler 1D PDEs system.

Appendix B.1. Euler 1D baseline API

The 1D Euler PDEs system is a non linear, hyperbolic (inviscid) system of conservation laws for compressible gas dynamics, that reads

$$U_t = R(U) \Leftrightarrow U_t = F(U)_x$$
$$U = \begin{bmatrix} \rho \\ \rho u \\ \rho E \end{bmatrix} \quad F(U) = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho u H \end{bmatrix} \tag{B.1}$$

Table A.16: Williamson's table of low storage Runge-Kutta schemes

(a) 5 stages, 4^{th} order

Stage	A	B	C
1	0	$\frac{1432997174477}{9575080441755}$	0
2	$-\frac{567301805773}{1357537059087}$	$\frac{5161836677717}{13612068292357}$	$\frac{1432997174477}{9575080441755}$
3	$-\frac{2404267990393}{2016746695238}$	$\frac{1720146321549}{2090206949498}$	$\frac{2526269341429}{6820363962896}$
4	$-\frac{3550918686646}{2091501179385}$	$\frac{3134564353537}{4481467310338}$	$\frac{2006345519317}{3224310063776}$
5	$-\frac{1275806237668}{842570457699}$	$\frac{2277821191437}{14882151754819}$	$\frac{2802321613138}{2924317926251}$

(b) 6 stages, 4^{th} order

Stage	A	B	C
1	0	0.122000000000	0
2	−0.691750960670	0.477263056358	0.122000000000
3	−1.727127405211	0.381941220320	0.269115878630
4	−0.694890150986	0.447757195744	0.447717183551
5	−1.039942756197	0.498614246822	0.749979795490
6	−1.531977447611	0.186648570846	0.898555413085

(c) 7 stages, 4^{th} order

Stage	A	B	C
1	0.000000000000	0.117322146869	0.000000000000
2	−0.647900745934	0.503270262127	0.117322146869
3	−2.704760863204	0.233663281658	0.294523230758
4	−0.460080550118	0.283419634625	0.305658622131
5	−0.500581787785	0.540367414023	0.582864148403
6	−1.906532255913	0.371499414620	0.858664273599
7	−1.450000000000	0.136670099385	0.868664273599

(d) 12 stages, 4^{th} order

Stage	A	B	C
1	0.0000000000000000	0.0650008435125904	0.0000000000000000
2	−0.0923311242368072	0.0161459902249842	0.0650008435125904
3	−0.9441056581158819	0.5758627178358159	0.0796560563081853
4	−4.3271273247576394	0.1649758848361671	0.1620416710085376
5	−2.1557771329026072	0.3934619494248182	0.2248877362907778
6	−0.9770727190189062	0.0443509641602719	0.2952293985641261
7	−0.7581835342571139	0.2074504268408778	0.3318332506149405
8	−1.7977525470825499	0.6914247433015102	0.4094724050198658
9	−2.6915667972700770	0.3766646883450449	0.6356954475753369
10	−4.6466798960268143	0.0757190350155483	0.6806551557645497
11	−0.1539613783825189	0.2027862031054088	0.7143773712418350
12	−0.5943293901830616	0.2167029365631842	0.9032588871651854

(e) 13 stages, 4^{th} order

Stage	A	B	C
1	0.0000000000000000	0.0271990297818803	0.0000000000000000
2	−0.6160178650170565	0.1772488819905108	0.0271990297818803
3	−0.4449487060774118	0.0378528418949694	0.0952594339119365
4	−1.0952033345276178	0.6086431830142991	0.1266450286591127
5	−1.2256030785959187	0.2154313974316100	0.1825883045699772
6	−0.2740182222332805	0.2066152563885843	0.3737511439063931
7	−0.0411952089052647	0.0415864076069797	0.5301279418422206
8	−0.1797084899153560	0.0219891884310925	0.5704177433952291
9	−1.1771530652064288	0.9893081222650993	0.5885784947099155
10	−0.4078831463120878	0.0063199019859826	0.6160769826246714
11	−0.8295636426191777	0.3749640721105318	0.6223252334314046
12	−4.7895970584252288	1.6080235151003195	0.6897593128753419
13	−0.6606671432964504	0.0961209123818189	0.9126827615920843

(f) 14 stages, 4^{th} order

Stage	A	B	C
1	0.0000000000000000	0.0367762454319673	0.0000000000000000
2	−0.7188012108672410	0.3136296607553959	0.0367762454319673
3	−0.7785331173421570	0.1531848691869027	0.1249685262725025
4	−0.0053282796654044	0.0030097086818182	0.2446177702277698
5	−0.8552979934029281	0.3326293790646110	0.2476149531070420
6	−3.9564138245774565	0.2440251405350864	0.2969311120382472
7	−1.5780575380587385	0.3718879239592277	0.3978149645802642
8	−2.0837094552574054	0.6204126221582444	0.5270854589440328
9	−0.7483334182761610	0.1524043173028741	0.6981269994175695
10	−0.7032861106563359	0.0760894927419266	0.8190890835352128
11	0.0013917096117681	0.0077604214040978	0.8527059887098624
12	−0.0932075369637460	0.0024647284755382	0.8604711817462826
13	−0.9514200470875948	0.0780348340049386	0.8627060376969976
14	−7.1151571693922548	5.5059777270269628	0.8734213127600976

Table A.17: Implicit Adams-Moulton coefficients

N_s	b_0	b_1	b_2	b_3
1	$\frac{1}{2}$	$\frac{1}{2}$	/	/
2	$-\frac{1}{12}$	$\frac{8}{12}$	$\frac{5}{12}$	/
3	$\frac{1}{24}$	$-\frac{5}{24}$	$\frac{19}{24}$	$\frac{9}{24}$

where ρ is the density, u is the velocity, p the pressure, E the total internal specific energy and H the total specific enthalpy. The PDEs system must completed with the proper initial and boundary conditions. Moreover, an ideal (thermally and calorically perfect) gas is considered

$$\begin{aligned} R &= c_p - c_v \\ \gamma &= \tfrac{c_p}{c_v} \\ e &= c_v T \\ h &= c_p T \end{aligned} \tag{B.2}$$

where R is the gas constant, c_p and c_v are the specific heats at constant pressure and volume (respectively), e is the internal energy, h is the internal enthalpy and *T* is the temperature. The following addition equations of state hold:

$$\begin{aligned} T &= \frac{p}{\rho R} \\ E &= \rho e + \frac{1}{2}\rho u^2 \\ H &= \rho h + \frac{1}{2}\rho u^2 \\ a &= \sqrt{\frac{\gamma p}{\rho}} \end{aligned} \tag{B.3}$$

An extension of the above Euler system is considered allowing the modelling of a multi-fluid mixture of different gas (with different physical characteristics). The well known Standard Thermodynamic Model is used to model the gas mixture replacing the density with the density fraction of each specie composing the mixture. This led to the following system:

$$\begin{gathered} U_t = R(U) \Leftrightarrow U_t = F(U)_x \\ U = \begin{bmatrix} \rho_s \\ \rho u \\ \rho E \end{bmatrix} \quad F(U) = \begin{bmatrix} \rho_s u \\ \rho u^2 + p \\ \rho u H \end{bmatrix} \quad for\ s = 1, 2, ...N_s \\ \rho = \sum_{s=1}^{N_s} \rho_s \\ c_p = \sum_{s=1}^{N_S} \frac{\rho_s}{\rho} c_{p,s} \quad c_v = \sum_{s=1}^{N_S} \frac{\rho_s}{\rho} c_{v,s} \end{gathered} \tag{B.4}$$

where N_s is the number of initial species composing the gas mixture.

Appendix B.1.1. Memory organization

The finite volume, Godunov's like approach is employed. Essentially, the method of Lines is used to decouple the space operator from the time one. Firstly, the space operator (the residual function of equation B.1) is numerically solved in order to reduce the original PDEs system to a system of ODEs that is then integrated over time by means of FOODIE solvers. Here we omit the details of the numerical models, interested readers can see []. On the contrary, some details on the memory organization is necessary to explaining the implemented API.

The conservative variables are co-located at the cell center. The cell and (inter)faces numeration is as shown in listing 40.

```
         cell    (inter)faces
          |              |
          v              v
|----|----|-...-|----|---|---|---|-...-|-----|-----|-----|-...-|-------|-------|
|1-Ng|2-Ng|  ...  |-1   |  0  |  1  |  2  |  ...  |   Ni  |  Ni+1 |  Ni+1 |  ...  |Ni+Ng-1|  Ni+Ng  |
|----|----|-...-|----|---|---|---|-...-|-----|-----|-----|-...-|-------|-------|
0-Ng                      -1    0    1    2      Ni-1    Ni                          Ni+Ng
```

Listing 40: Numerical grid organization

In listing 40 Ni is the number of finite volumes (cells) used for discretizing the domain and Ng is the number of ghost cells used for imposing the left and right boundary conditions (for a total of $2Ng$ cells). For each cell the conservative variables must be stored: this is done by means of of rank 2 array where the first index refers to the conservative variables (densities, momentum or energy) while the second index refers to the space location, namely the cell index.

The most CPU time consuming part of a finite volume scheme is the fluxes computation across the cells interfaces. Such a computation corresponds to a loop over all the cells interfaces. Listing 41 shows a pseudo-code example of such a computation.

```
do i=0, Ni
  F(:, i) = compute_fluxes(U(:, i), U(:, i+1))
enddo
```

Listing 41: Pseudo-code example of fluxes computation

In the pseudo-code of listing 41 it has been emphasized that the fluxes across an interface depends on the cells at left and right of the interface itself. The key point for the parallelization of such an algorithm is to compute the fluxes concurrently using as much as possible the parallel resources provided by the running architecture. As a consequence, the above showed loop over the whole domain is split into sub-domains (explicitly or implicitly accordingly to the parallel model adopted) and the fluxed of each sub-domain are computed concurrently.

Appendix B.1.2. The serial API

The conservative variables of 1D Euler's system can be easily implemented as a *FOODIE integrand field* defining a concrete extension of the FOODIE *integrand* type. Listing 42 reports the implementation of such an integrand field that is contained into the tests suite shipped within the FOODIE library.

```
type, extends(integrand) :: euler_1D
  private
  integer(I_P)                    :: ord=0       ! Space accuracy formal order.
  integer(I_P)                    :: Ni=0        ! Space dimension.
  integer(I_P)                    :: Ng=0        ! Number of ghost cells for boundary conditions handling.
  integer(I_P)                    :: Ns=0        ! Number of initial species.
  integer(I_P)                    :: Nc=0        ! Number of conservative variables, Ns+2.
  integer(I_P)                    :: Np=0        ! Number of primitive variables, Ns+4.
  real(R_P)                       :: Dx=0._R_P   ! Space step.
  type(weno_interpolator_upwind)  :: weno        ! WENO interpolator.
  real(R_P),      allocatable     :: U(:,:)      ! Integrand (state) variables, whole physical domain [1:Nc,1:Ni].
  real(R_P),      allocatable     :: cp0(:)      ! Specific heat cp of initial species [1:Ns].
  real(R_P),      allocatable     :: cv0(:)      ! Specific heat cv of initial species [1:Ns].
  character(:),   allocatable     :: BC_L        ! Left boundary condition type.
  character(:),   allocatable     :: BC_R        ! Right boundary condition type.
  integer(I_P)                    :: me=0        ! ID of this_image().
  integer(I_P)                    :: we=0        ! Number of CAF images used.
  contains
    ! auxiliary methods
    procedure, pass(self), public :: init
    procedure, pass(self), public :: destroy
    procedure, pass(self), public :: output
    procedure, pass(self), public :: dt => compute_dt
    ! ADT integrand deferred methods
    procedure, pass(self), public :: t => dEuler_dt
    procedure, pass(lhs),  public :: local_error => euler_local_error
    procedure, pass(lhs),  public :: integrand_multiply_integrand => euler_multiply_euler
    procedure, pass(lhs),  public :: integrand_multiply_real => euler_multiply_real
    procedure, pass(rhs),  public :: real_multiply_integrand => real_multiply_euler
    procedure, pass(lhs),  public :: add => add_euler
    procedure, pass(lhs),  public :: sub => sub_euler
    procedure, pass(lhs),  public :: assign_integrand => euler_assign_euler
    procedure, pass(lhs),  public :: assign_real => euler_assign_real
    ! private methods
    procedure, pass(self), private :: primitive2conservative
    procedure, pass(self), private :: conservative2primitive
    procedure, pass(self), private :: synchronize
    procedure, pass(self), private :: impose_boundary_conditions
    procedure, pass(self), private :: reconstruct_interfaces_states
    procedure, pass(self), private :: riemann_solver
    final                          :: finalize
endtype euler_1D
```

Listing 42: implementation of the *Euler 1D integrand* type

Appendix B.2. Euler 1D CAF Enabled API

Appendix B.3. Euler 1D OpenMP Enabled API

References